2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NEURAL NETWORKS APPLIED TO SIGNAL PROCESSING

by

Mark D. Baehre

September 1989

Thesis Advisor: Murali Tummala

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) 62	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.	
11. TITLE (Include Security Classification) NEURAL NETWORKS APPLIED TO SIGNAL PROCESSING			
12. PERSONAL AUTHOR(S) BAEHRE, Mark D.			
13a. TYPE OF REPORT Engineer's Thesis	13b TIME COVERED FROM ____ TO ____	14 DATE OF REPORT (Year, Month, Day) 1989 September	15 PAGE COUNT 107
16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Neural network, backpropagation, conjugate gradient method, Fibonacci line search, nonlinear signal processing, channel equalization, Theses, Artificial Intelligence, Computer Architecture

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The relationship between the structure of a neural network and its ability to perform nonlinear mapping is analyzed. A new algorithm, called the conjugate gradient optimization method, for calculating the weights and thresholds of a neural network is presented. The performance of the conjugate gradient algorithm is then compared to the well known backpropagation method and shown to be more computationally efficient. A neural network using the conjugate gradient algorithm is then applied to three simple examples to demonstrate its signal processing capabilities. The first example illustrates the ability of the neural network to perform classification. The second compares the performance of a one-step linear predictor to a neural network for a nonlinear chaotic time series. The neural network predictor is shown to provide much greater accuracy than its linear counterpart. The final application presented demonstrates the ability of a neural network to perform channel equalization for a nonminimum phase channel. Its performance is then compared to its linear equivalent. Keywords

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Murali Tummala	22b TELEPHONE (Include Area Code) 408-646-2645 / 22c OFFICE SYMBOL 62Tu

DD Form 1473, JUN 86 — *Previous editions are obsolete* — SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 — UNCLASSIFIED

Neural Networks Applied to Signal Processing

by

Mark D. Baehre
Captain, United States Army
B.S., United States Military Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL

September 1989

Author: [illegible]

Mark D. Baehre

Approved by: [illegible]

Murali Tummala, Thesis Advisor

Charles W. Therrien

Charles W. Therrien, Second Reader

[illegible]

John P. Powers, Chairman
Department of Electrical and Computer Engineering

[illegible]

Gordon E. Schacher
Dean of Science and Engineering

ABSTRACT

The relationship between the structure of a neural network and its ability to perform nonlinear mapping is analyzed. A new algorithm, called the conjugate gradient optimization method, for calculating the weights and thresholds of a neural network is presented. The performance of the conjugate gradient algorithm is then compared to the well known backpropagation method and shown to be more computationally efficient. A neural network using the conjugate gradient algorithm is then applied to three simple examples to demonstrate its signal processing capabilities. The first example illustrates the ability of the neural network to perform classification. The second compares the performance of a one-step linear predictor to a neural network for a nonlinear chaotic time series. The neural network predictor is shown to provide much greater accuracy than its linear counterpart. The final application presented demonstrates the ability of a neural network to perform channel equalization for a nonminimum phase channel. Its performance is then compared to its linear equivalent.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

The completion of this thesis can be attributed to the inspiration, knowledge, and support of a countless number of people. I would like to specifically thank Professor Murali Tummala, my thesis advisor and Professor Charles W. Therrien, my second reader, for sharing their expertise, providing encouragement, and demonstrating unfailing patience during the preparation of this thesis.

I dedicate this thesis to my wife, Sheila. Her understanding, support, and love make everything worthwhile.

I. INTRODUCTION

Artificial neural networks have been studied for many years in the hope of achieving human–like performance. Neural networks consist of highly connected sets of relatively simple processing elements. Computations are performed collectively by the entire network with the activity distributed over all the processing elements. This parallel distributed processing provides neural networks with the potential to solve complex problems more quickly than the currently well known present serial methods. The nonlinear nature and simple structure of neural networks provide a formalism for the study of nonlinear signal processing.

The application of neural networks to signal processing involves developing an understanding of the relationship between the structure of a neural network and its ability to perform the desired input–to–output mapping. A neural network's structure is defined by the number and type of processing elements in the network, the values of the weights that connect the processing elements together, and a threshold value associated with each processing element. Past work has lead to a large variety of neural network models. The models include the *Hopfield network*, the *single-* and *multi-layer perceptron networks*, the *reduced Coulomb energy (RCE) classifier*, and the *adaptive resonance theory (ART) model* [Ref. 1:pp. 65–73]. Each model differs in its structure and the manner in which the weights and thresholds of the network are derived. One current method for calculating the weights and thresholds of a *feedforward* multilayer neural network, called the *backpropagation* method, uses a steepest descent method to iteratively adapt the weights and thresholds of the network [Ref. 2:p. 127]. This method has generally been shown to be slow to converge to the optimal set of weights and thresholds for a given problem [Ref. 1:p. 300]. The objectives of this thesis research were therefore:

- Investigate the relationship between the structure of a neural network and its ability to perform input–output mapping.
- Develop an alternative to the backpropagation method that converges more quickly to the optimal set of weights and thresholds for any given problem.
- Compare the performance of a neural network to its linear counterpart for some representative signal processing applications.

Chapter II provides a general overview of the theory of neural networks. A graphical approach is employed to demonstrate the ability of neural networks to perform nonlinear mapping for various network configurations. The results are then related to a theorem by Kolmogorov. The backpropagation method for calculating the weights and thresholds of the neural network is also introduced.

Chapter III deals with the derivation of an alternative algorithm to the backpropagation method for calculating the weights and thresholds of a neural network. The conjugate gradient optimization method is presented and then applied to the neural network model. The Fibonacci line search method used in conjunction with the conjugate gradient method is also discussed. The final section of the chapter presents details concerning actual implementation of the algorithm to include experimentally derived parameters.

Chapter IV presents the results of the thesis research. The conjugate gradient algorithm's performance is compared to the backpropagation method and is shown to be more computationally efficient. A neural network using the conjugate gradient algorithm is then applied to three simple examples to validate the performance of the new algorithm and to demonstrate the types of tasks that a neural network can perform. The first example illustrates the neural network's ability to perform classification. A two input neural network is successfully "taught" to differentiate between sets of points falling inside and outside a circle. The second example compares the performance of a one-step linear predictor to a neural network for a nonlinear chaotic time series generated using the Feigenbaum logistic function. This application demonstrates the nonlinear mapping ability of the neural network. The neural network predictor is shown to provide much greater accuracy than its linear counterpart. The final application presented demonstrates the ability of a neural network to perform channel equalization for a nonminimum phase channel. Its performance is compared to its linear equivalent and is shown to provide superior performance.

Chapter V contains the overall conclusions of the thesis research and provides recommendations for future research.

II. FUNDAMENTALS - HOW NEURAL NETWORKS WORK

A. THE BASIC BUILDING BLOCK

A neural network is a system of relatively simple processing elements whose function is determined by its network structure, connection weights, and the transfer function of each neuron. Figure 2.1 shows a single artificial neuron, the fundamental building block for all neural networks. A set of inputs $x_1, x_2, \ldots, x_n$ are applied through a set of associated connection weights $w_1, w_2, \ldots, w_n$ to the neuron.

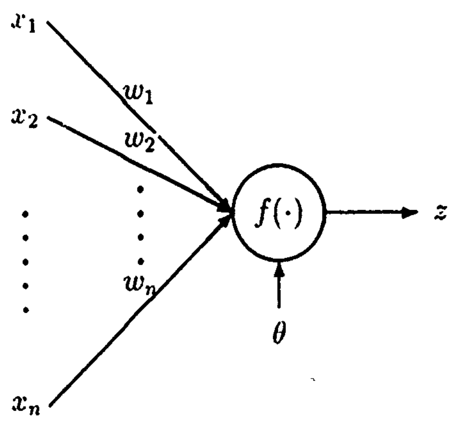

Figure 2.1: A single artificial neuron

The inputs correspond to the stimulation levels and the weights to the synaptic strengths of a biological neuron. The neuron sums the weighted inputs, adds a threshold value, and applies the result to the neuron's transfer function $f(\mathbf{x})$. This operation can be expressed as

$$z = f\left(\sum_i w_i x_i + \theta\right) \qquad (2.1)$$

or in vector notation

$$z = f\left(\mathbf{w}^T\mathbf{x} + \theta\right) \tag{2.2}$$

where $\mathbf{x}$ is a column vector of inputs, $\mathbf{w}$ the corresponding column vector of weights, and θ the neuron's threshold value.

B. THE TRANSFER FUNCTION

A number of possibilities arise for selection of an appropriate transfer function. These include most notably: the signum function, the linear function, and the sigmoid function. Initial research conducted in the 1950's and 1960's by Rosenblat, Minsky and others used the signum function shown in Figure 2.2 [Ref. 3]. The signum function will be used for a preliminary discussion of how neural networks operate.

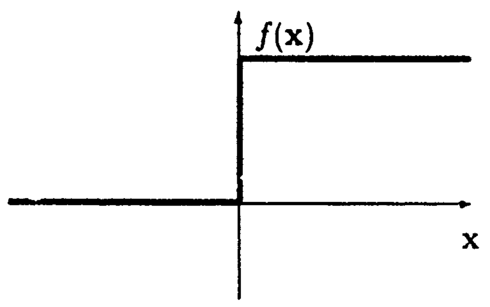

Figure 2.2: Signum function

Artificial neurons using the signum transfer function were referred to as perceptrons [Ref. 3]. The signum transfer function causes the output of the perceptron to take one of two discrete values. The point at which the neuron switches from low to high or high to low is determined by the input weights and the perceptron's threshold value. It has been shown that a single perceptron has the ability to distinguish between two classes of inputs [Ref. 4:p. 13]. This is demonstrated in Figure 2.3 for a two input network.

The combination of weights (w_1 and w_2) and the offset (θ) define a line where the output of the network (z) is high for the class of inputs falling on one side of the line and low for the second class of inputs falling on the other side. If there are n inputs to a single perceptron, as pictured in Figure 2.1, the perceptron can construct an n dimensional hyperplane separating the two classes of inputs. Input classes that cannot be separated by a simple hyperplane therefore cannot be accurately differentiated by a single perceptron.

This problem can be remedied by cascading the perceptrons into several layers. This type of network topology is called a feedforward network because the output from the previous layer is fed forward to only the neurons in the next layer of the network. By adding additional layers, more complex boundaries can be defined. A two layer network is capable of defining decision regions that are convex or concave in shape. For the two input case shown in Figure 2.4, each perceptron in the first layer defines a boundary line. A single second layer perceptron weights and combines the outputs from the first layer perceptrons to produce the two decision regions. As pictured in Figure 2.4 a two layer network can also define a single enclosed region. With the addition of a third layer, disjoint enclosed regions can be combined to create a decision map of any arbitrary complexity, given a sufficient number of perceptrons in each layer. This is illustrated in Figure 2.5.

The performance of a multilayer perceptron network using the signum transfer function is satisfactory provided the desired output from the network is limited to two discrete values (i.e., high or low). This would be appropriate for a binary classifier system, where each output would represent one of two classes, i.e., a binary value. It does not, however, provide sufficient resolution for analog (continuously valued) or the corresponding discrete valued output functions associated with most other signal processing applications.

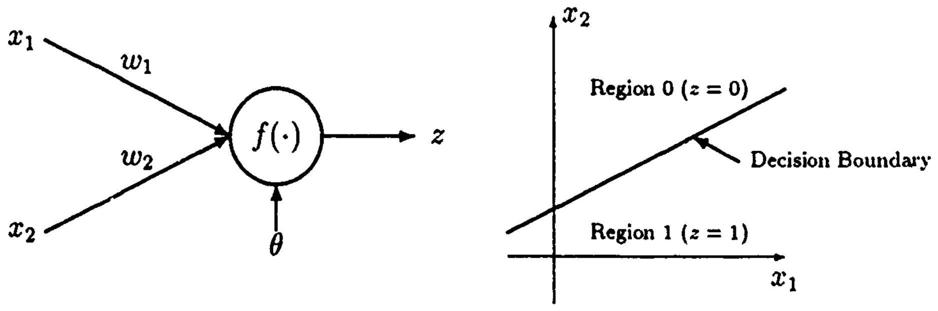

Figure 2.3: Single neuron and associated decision regions

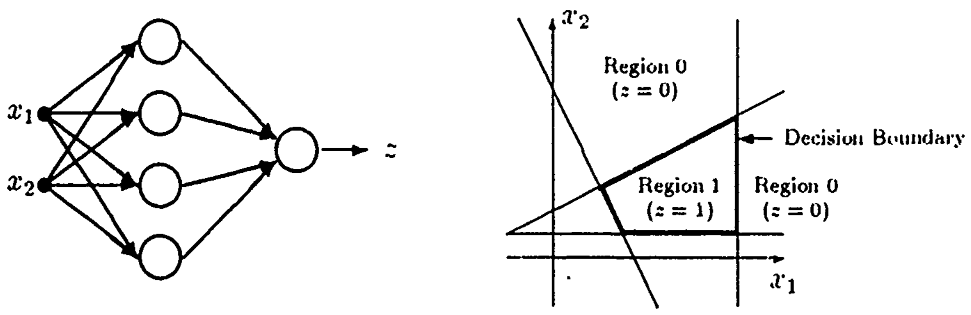

Figure 2.4: Two layer network and associated decision regions

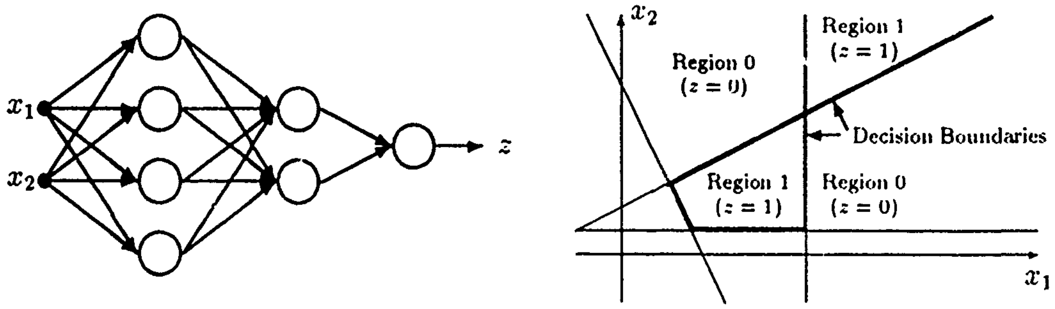

Figure 2.5: Three layer network and associated decision regions

One example of a transfer function that would be capable of providing such a continuously variable output is the linear transfer function. In this case, the output of the artificial neuron would simply be the weighted sum of the inputs plus the neuron's threshold value. This can be expressed as

$$z = f(\mathbf{x}) = \sum_i w_i x_i + \theta \tag{2.3}$$

or in vector notation

$$z = f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + \theta. \tag{2.4}$$

This is the transfer function used by Widrow and Hoff in their development of the adaptive linear (adaline) and multiple adaptive linear (madaline) filters [Ref. 5:p. 10]. A great deal has been written concerning research and applications of the adaptive linear filter although it has not often been referred to as a neural model [Ref. 6],[Ref. 7],[Ref. 8]. One key feature of the linear neural network is that there is no functional difference between a multilayer and a single layer network. For example, for the simple two layer network in Figure 2.6 the output of the first layer neurons can be written as

$$f_1(x_1, x_2) = w_1 x_1 + w_2 x_2 + \theta_1 \tag{2.5}$$

and

$$f_2(x_1, x_2) = w_3 x_1 + w_4 x_2 + \theta_2. \tag{2.6}$$

The output of the network can then be written as

$$f_3(x_1, x_2) = w_5 f_1(x_1, x_2) + w_6 f_2(x_1, x_2) + \theta_3. \tag{2.7}$$

After some algebraic manipulation and substitution, the final result is

$$f_3(x_1, x_2) = w_5(w_1 + w_3)x_1 + w_6(w_2 + w_4)x_2 + (w_5\theta_1 + w_6\theta_2 + \theta_3). \tag{2.8}$$

From the above discussion, it is clear that, regardless of the number of layers in the network, the network can always be reduced to a single layer network. Essentially then, the linear adaptive filter is nothing more than the linear version of a single layer neural network.

A third transfer function which has been recently popularized by Rumelhart et al. [Ref. 9] is called the sigmoid function. It is defined by the equation

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.9}$$

where

$$z = \sum_i w_i x_i + \theta = \mathbf{w}^{\mathbf{T}}\mathbf{x} + \theta. \tag{2.10}$$

The sigmoid function, pictured in Figure 2.7, has a shape which would appear to fall somewhere between the linear transfer function and the signum transfer function. Its output is limited to a continuous range of values between zero and one. For values of z near zero. the transfer function behaves in a linear fashion with a constant slope of one. If the input weights to the neuron are kept sufficiently small and the range of input values limited, the sigmoidal artificial neuron can be made to appear linear. Likewise, by using large values for the input weights w, the values for z would vary more rapidly and the sigmoidal artificial neuron would more closely approximate the signum function. As a result, the output of the network can be made to approximate both linear and nonlinear combinations of the inputs depending on the values of the network's weights ($\mathbf{w}$) and thresholds (θ).

A theorem developed by Kolmogorov and described in Reference 10 provides further insight into the potential capabilities of a multilayer sigmoidal neural network. The theorem states that any continuous function of n variables can be represented using only linear summations and nonlinear but continuously increasing functions of only one variable. This would indicate that a three layer artificial neuron feedforward

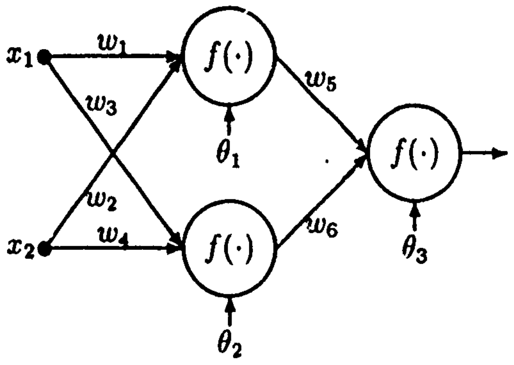

Figure 2.6: Two layer linear network

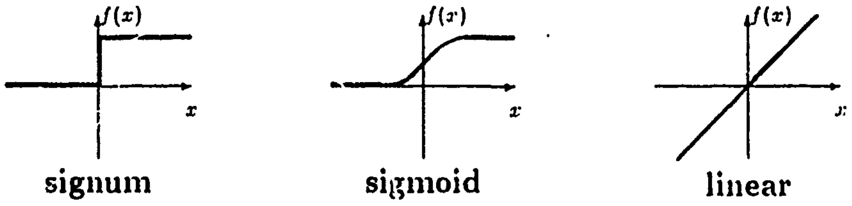

Figure 2.7: Neuron transfer functions

network using a sigmoidal transfer function is capable of representing any nonlinear multivariable function. The theorem, however, does not indicate the number of neurons required in each layer, or how the values for the weights should be derived.

It has been suggested that one approach to representing an n-dimensional nonlinear function using neural networks might be by a weighted combination of n-dimensional 'bumps' [Ref. 11]. This is somewhat analogous to the Fourier series representation of an arbitrary signal where weighted combinations of sinusoids of suitable frequencies are used. To see how a nonlinear function might be represented using a sigmoidal neural network, let us look at the case where we have a nonlinear function of two variables. The output of the nonlinear function could be interpreted as a two dimensional surface in a three dimensional space. The output of a single sigmoidal neuron would have a surface like that pictured in Figure 2.8.

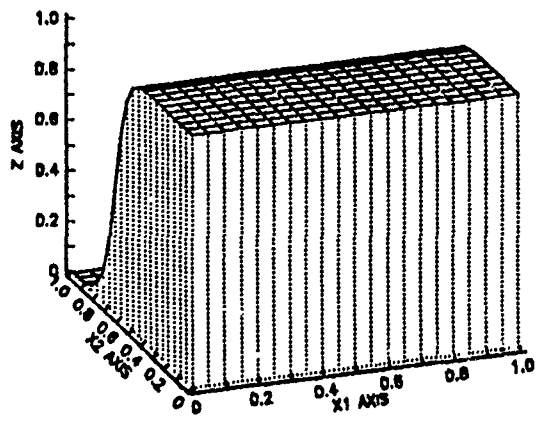

Figure 2.8: A sigmoid surface

The orientation of the rising slope of the sigmoidal surface is determined by the neuron's input weights (w). Its position is determined by its threshold (θ) value. The height of the surface is controlled by the weight connected to the output of the neuron. If we add a second neuron with the same orientation, but a slightly different position than the first by using a different threshold value (θ), and use an output weight equal to but opposite in sign of the first, we can form a ridge as shown in Figure 2.9.

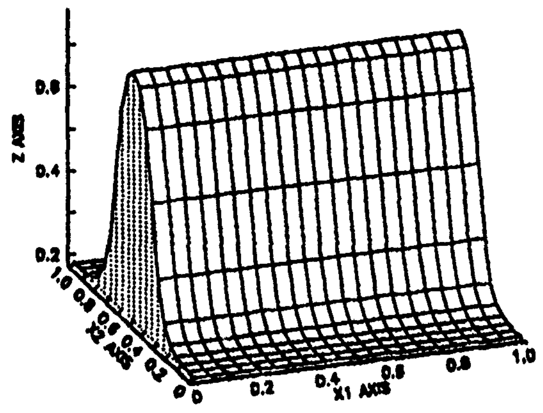

Figure 2.9: A ridge

A second ridge, perpendicular to the first, can then be constructed by adding two additional neurons to the first layer and selecting appropriate input weight values. The sum of the two ridges then forms the surface pictured in Figure 2.10. The weights connecting the outputs of the first layer neurons to the single second layer neuron along with the second layer neuron's threshold value can then be adjusted to yield a true bump shown in Figure 2.11.

We can now represent any surface as a combination of these bumps. The network topology to accomplish this would consist of multiple copies of two layer network and a single third layer neuron to weight and sum the bumps. The resulting surface is pictured in Figure 2.12. The preceding development provides some insight into the number of neurons required in each layer of a neural network to adequately represent

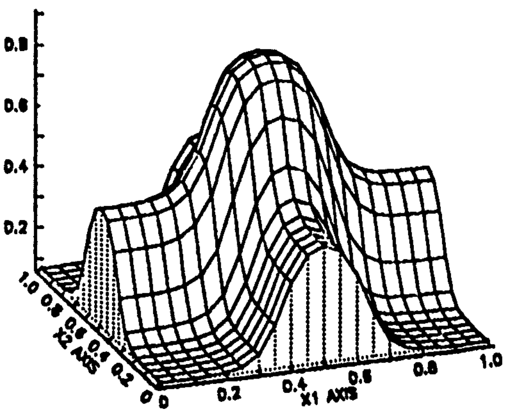

Figure 2.10: A pseudo-bump

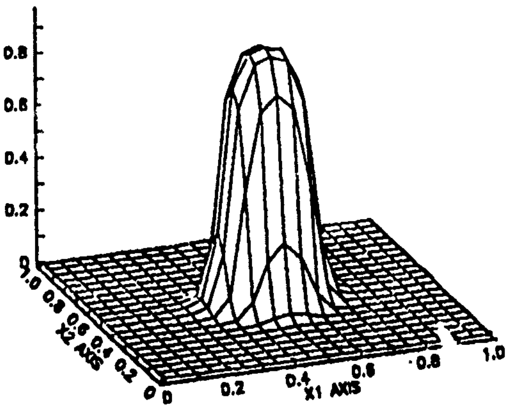

Figure 2.11: A bump

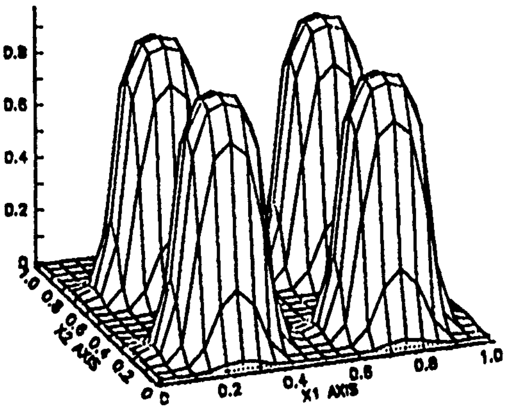

Figure 2.12: Multiple bumps

a given nonlinear function. A given function might be more efficiently represented using a combination of sigmoidal surfaces or ridges rather than bumps. The better knowledge one has of the function to be represented will lead to a better decision concerning the neural network topology required.

C. CALCULATION OF WEIGHTS AND THRESHOLDS

The burning question that has yet to be addressed concerning the feedforward sigmoidal neural network is how do we calculate the weights (**w**) and the neuron thresholds (θ) of the network to yield a satisfactory representation of a given nonlinear function. A method called backpropagation, developed by Rumelhart, has proven popular and has been demonstrated to work fairly well [Ref. 2]. The backpropagation method uses a training data set consisting of sets of inputs and a desired output value. A set of inputs is applied to the neural network and the resulting network output is compared to the desired value. The error between the neural network's output and the desired output, along with the current state of neural network, is used to modify the neural network's weights and threshold values. The state of the neural network is defined by the current input to the network, its weights, thresholds, and each neuron's transfer function. The backpropagation method attempts to minimize the sum of the squared errors over the entire training data set. This can be expressed as

$$E = \sum_t e^2(t) = \sum_t (y(t) - z(t))^2 \tag{2.11}$$

where E is the total squared error, $e(t)$ is the network output error for the t^{th} input set, $y(t)$ is the desired or target output for the t^{th} input set, and $z(t)$ is the actual output of the neural net for the t^{th} input set. The weights and the thresholds of the network are iteratively updated in proportion to the gradient of the total squared error, E. This can be expressed as

$$w(n+1) = w(n) + \Delta w(n) = w(n) - \frac{\delta E}{\delta w(n)} \cdot \epsilon \tag{2.12}$$

and

$$\theta(n+1) = \theta(n) + \Delta\theta(n) = \theta(n) - \frac{\delta E}{\delta\theta(n)} \cdot \epsilon \tag{2.13}$$

where $w(n)$ and $\theta(n)$ are the weights and thresholds at the n^{th} iteration of the algorithm, $\Delta w(n)$ and $\Delta\theta(n)$ are the incremental changes to the weights and thresholds, and ϵ is the proportionality constant [Ref. 2:p. 130]. The backpropagation method gets its name from the fact that the error at the output of the network is propagated back through the network in the form of gradients in order to update the network's weights and thresholds.

The backpropagation method is essentially a steepest descent optimization algorithm which uses the gradient of the squared error function to modify the weights and thresholds of the neural network [Ref. 2:p. 127]. One requirement dictated by this gradient method is that the transfer function of the neurons be continuously differentiable [Ref. 2:p. 131]. As a result, this method cannot be used with the signum transfer function because of its discontinuity. The method, however. does work for the linear and sigmoidal transfer function cases.

As presented above, the weights and thresholds are updated after a complete pass of the entire training data set through the network. In the actual implementation of the algorithm, however, Rumelhart updates the weights and thresholds of the network after each input/desired output pair is applied [Ref. 2:pp. 136-137]. His rationale for doing this is that the algorithm converges so slowly that it does not affect the overall convergence rate, and that it is more gratifying to update the weights and thresholds more frequently [Ref. 2:p. 137]. As Rummelhart indicated, the steepest descent method is extremely slow to converge. It was this deficiency that led to the development of this thesis project. Lapedes and Farber indicated that a related optimization method, the conjugate gradient algorithm, yielded a significant improvement in the convergence rate of the backpropagation method [Ref. 12]. The following chapter will address the development and application of this optimization method to a feedforward sigmoidal neural network.

III. DERIVATION OF THE ADAPTATION ALGORITHM

A. THE CONJUGATE GRADIENT METHOD

1. General Description

The conjugate gradient method is an iterative method for optimizing a set of coefficients **h** in order to minimize a given objective function $J(\mathbf{h})$. It falls into the class of optimization methods that apply a multidimensional search using derivatives to the optimization problem [Ref. 13:pp. 289–316]. The steepest descent method, which Rumelhart uses for adapting the feedforward neural network, is also a member of this class [Ref. 2]. This class of optimization methods, called gradient methods, treat the objective function $J(\mathbf{h})$ as a multidimensional surface over which it iteratively searches for the absolute or global minimum [Ref. 13:pp. 289–316]. The coefficients **h** are the multidimensional coordinates which define where the algorithm is located on the surface during any particular iteration. This class of optimization methods require that the objective function be differentiable with respect to the coefficients **h** that are adapted [Ref. 13:p. 289]. This partial derivative is called the gradient **g** of the objective function. When evaluated for a given set of coefficients **h**, the gradient **g** is a multidimensional vector which is tangent to the objective function surface at a point defined by the coefficients **h**. This vector points in the direction of greatest increase. The negative of the gradient $(-\mathbf{g})$ logically points downhill in the direction of greatest decrease. Thus, the gradient vector **g** can provide a direction along the surface of the objective function in which to search for the global minimum. The advantage of gradient methods is that they decompose the optimization problem from a multidimensional search of the objective function surface to a sequence of line searches along directions determined by the gradient vector **g**.

The method of steepest descent uses the gradient vector **g** directly to perform its iterative line search of the objective function surface [Ref. 14:pp. 214–220]. Rumelhart points out that the steepest descent method works well when the objective function surface is quadratic or bowl–shaped with a single global minimum [Ref. 2:p. 132]. He states, however, that the more complex objective function surfaces associated with multilayer neural networks frequently contain many local minima [Ref. 2:p. 132]. As a result, the steepest descent method can become trapped in one of these local minima yielding a less than optimal solution. This is because the magnitude of gradient vector decreases as the algorithm approaches a local minimum. The distance the steepest descent algorithm travels for a given iteration is a function of a constant times the magnitude of the gradient. Therefore, as the magnitude of the gradient decreases, the distance the algorithm travels along the surface decreases. Compounding the problem is the fact that each successive gradient is orthogonal to the previous gradient. This causes the algorithm to zigzag in ever smaller steps as it approaches the bottom of a local minimum. The result is that the algorithm becomes trapped at the bottom of a local minimum and never reaches the optimal point or global minimum. Use of a constant stepsize also causes the steepest descent algorithm to be extremely slow to converge [Ref. 13:pp. 290–291].

The conjugate gradient approach is motivated by a desire to accelerate the convergence rate of the steepest descent method without greatly increasing the complexity of the algorithm. The conjugate gradient method uses a succession of direction vectors $\mathbf{d}_k$ that are conjugate to the gradient vector $\mathbf{g}_k$ obtained as the algorithm progresses. The direction along which the algorithm searches, $\mathbf{d}_k$, is a linear combination of present and past values of the gradient vector. The result is that the gradient vector $\mathbf{g}_k$ is orthogonal to the subspace Γ_k which is defined by the set of all previous direction vectors $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k-1}$. Each successive iteration essentially adds an additional dimension to the subspace Γ_k. The distance α_k that the algorithm travels along the line search direction $\mathbf{d}_k$ also varies for each iteration of the algorithm. This makes the method only slightly more complicated than the steepest descent method. The algorithm, however, does not become trapped in local minima as easily as the steepest descent method and converges steadily to the global minimum or optimal set of coefficients $\mathbf{h}_k$ [Ref. 13:pp. 297–316].

2. Notation Summary

The notation used to describe the conjugate gradient method is as follows:

$J(\mathbf{h})$	Objective function to be minimized.
$\mathbf{h}_k$	Coefficient vector at the k^{th} iteration.
$\mathbf{g}_k$	Gradient vector of the objective function at the k^{th} iteration.
$\mathbf{d}_k$	Search direction vector at the k^{th} iteration.
α_k	Search distance coefficient at the k^{th} iteration.
β_k	Deflection coefficient at the k^{th} iteration.

3. Summary of the Conjugate Gradient Algorithm

A summary of the conjugate gradient method for minimizing a differentiable objective function $J(\mathbf{h})$ is listed below [Ref. 13:p. 306]:

Step 1. Choose an initial set of coefficients $\mathbf{h}_0$.

Step 2. Calculate the initial gradient $\mathbf{g}_0$ using the definition

$$\mathbf{g}_0 = \frac{\partial J(\mathbf{h}_0)}{\partial \mathbf{h}_0}. \tag{3.1}$$

Step 3. Let the initial direction vector be $\mathbf{d}_0 = -\mathbf{g}_0$.

Step 4. Let k=0.

Step 5. Let α_k be the optimal solution to the problem to minimize $J(\mathbf{h}_k + \alpha_k \mathbf{d}_k)$ subject to $\alpha_k \geq 0$.

Step 6. Update the new coefficients $\mathbf{h}_{k+1}$ using the equation

$$\mathbf{h}_{k+1} = \mathbf{h}_k + \alpha_k \mathbf{d}_k. \tag{3.2}$$

Step 7. Calculate the next gradient vector value $\mathbf{g}_{k+1}$ using the new coefficients $\mathbf{h}_{k+1}$.

Step 8. Calculate the deflection coefficient β_k using the equation

$$\beta_k = \frac{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}. \tag{3.3}$$

Step 9. Update the direction vector $\mathbf{d}_{k+1}$ using the equation

$$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k. \tag{3.4}$$

Step 10. Replace k by $k+1$ and go to step 5.

4. Selection of a Line Search Method

The conjugate gradient method outlined above requires that a search distance coefficient α_k be found that minimizes the objective function $J(\mathbf{h}_k + \alpha_k \mathbf{d}_k)$ subject to $\alpha_k \geq 0$. This dictates that a line search be performed starting at the point in multidimensional space defined by the current coefficient vector $\mathbf{h}_k$ and proceeding along the line defined by the current direction vector $\mathbf{d}_k$ until the minimum value of the objective function is found. The distance the line search algorithm travels from the point $\mathbf{h}_k$ to the minimum value of the function is then defined to be the scalar value α_k. A number of methods have been proposed to perform this line search. These include the uniform search, dichotomous search, the golden section method, and the Fibonacci method [Ref. 13:pp. 253–264]. There is also a class of line search methods which use derivatives to assist in finding the minimum value of the objective function [Ref. 13:pp. 264–269]. This second group of methods was considered for use with the conjugate gradient method but were subsequently rejected due to the complexity of calculating and evaluating the required derivatives. The selection of an appropriate line search method for use in conjunction with the conjugate gradient method was based primarily on efficiency. All of the methods except for the Fibonacci search require two evaluations of the objective function during each iteration of the algorithm. The Fibonacci method, however, requires only a single evaluation because it also uses the results from the previous iteration. Comparison of the line search methods mentioned above revealed that the Fibonacci search method is the most efficient [Ref. 13:p. 264]. As a result, the Fibonacci search method was chosen to be used in conjunction with the conjugate gradient method.

The Fibonacci method performs a search for the minimum value of a function of a single variable over a closed bounded interval $[a, b]$. The function in this case is $J(\mathbf{h}_k + \alpha_{kj}\mathbf{d}_k)$ where α_{kj} is the single variable. The interval over which the algorithm searches is called the interval of uncertainty and limits the range of values for α_{kj}. The lower limit for α_{kj} is given by the conjugate gradient method as zero, but the upper limit must be specified before the algorithm can begin. The interval of uncertainty is steadily reduced as the algorithm progresses. The number of iterations which the algorithm will perform must also be specified before the start of the algorithm. The Fibonacci method is based on the Fibonacci sequence F_v which is defined as

$$F_{v+1} = F_v + F_{v-1} \tag{3.5}$$

$$F_0 = F_1 = 1 \tag{3.6}$$

The resulting sequence is $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$. The Fibonacci search method begins by evaluating the objective function at each of two points within the interval of uncertainty as shown in figure 3.1.

These two points, which we will call λ_j and μ_j, are calculated using

$$\lambda_j = a_j + \frac{F_{n-j-1}}{F_{n-j+1}}(b_j - a_j) \tag{3.7}$$

and

$$\mu_j = a_j + \frac{F_{n-j}}{F_{n-j+1}}(b_j - a_j) \tag{3.8}$$

where k is the iteration index of the conjugate gradient algorithm, j is the iteration index of the Fibonacci algorithm, $[a_j, b_j]$ is the current interval of uncertainty, and n is the total number of iterations planned. A new interval of uncertainty $[a_{j+1}, b_{j+1}]$ is then selected based on the value of the objective function at the two points λ_j and μ_j. If $J(\mathbf{h}_k + \lambda_j \mathbf{d}_k) > J(\mathbf{h}_k + \mu_j \mathbf{d}_k)$, then the new interval of uncertainty $[a_{j+1}, b_{j+1}]$ is given by $[\lambda_j, b_j]$. Likewise, if the opposite is true, $J(\mathbf{h}_k + \lambda_j \mathbf{d}_k) < J(\mathbf{h}_k + \mu_j \mathbf{d}_k)$, then the new interval of uncertainty is $[a_j, \mu_j]$. Both cases are shown in Figure 3.2. The key feature that makes the Fibonacci method so attractive is that, for the next iteration $j + 1$, either $\lambda_{j+1} = \mu_j$ or $\mu_{j+1} = \lambda_j$, depending on which new interval of uncertainty was selected. Since the objective function has already been evaluated at the previous values for λ_j and μ_j, then only one additional evaluation must be made for each succeeding iteration. At the completion of the specified n iterations of the algorithm, the size of the final interval of uncertainty will be

$$(b_n - a_n) = \frac{(b_0 - a_0)}{F_n} \tag{3.9}$$

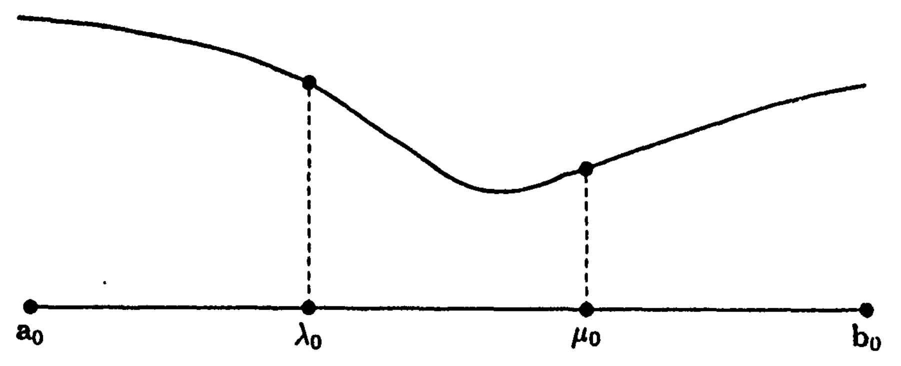

Figure 3.1: Initial evaluation points λ_0 and μ_0 and interval of uncertainty

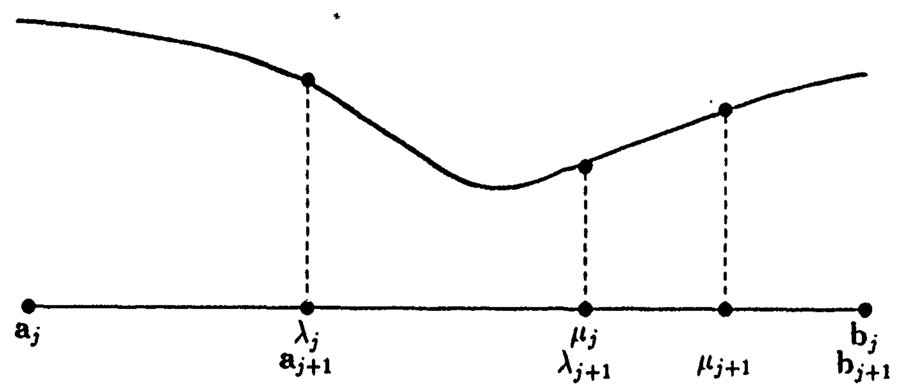

Figure 3.2: Evaluation points λ_{j+1} and μ_{j+1} and revised interval of uncertainty when $J(\lambda_j) > J(\mu_j)$

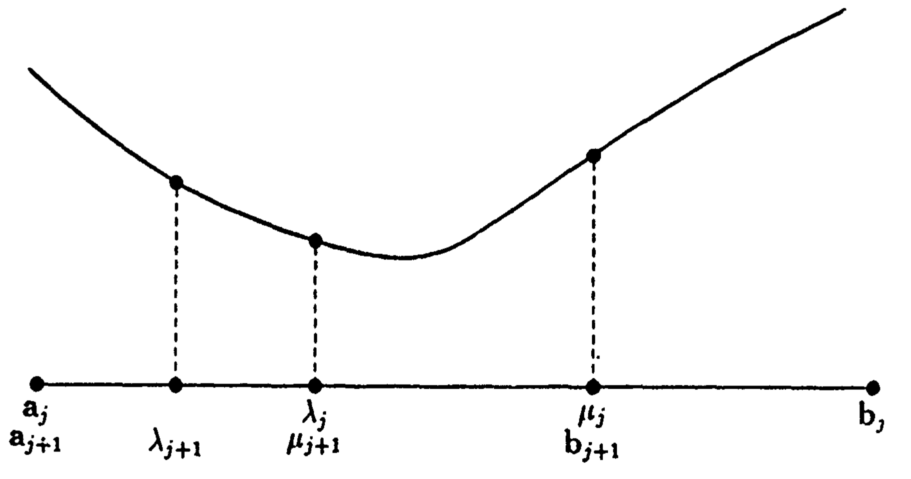

Figure 3.3: Evaluation points λ_{j+1} and μ_{j+1} and revised interval of uncertainty when $J(\lambda_j) < J(\mu_j)$

If we select the midpoint of the final interval of uncertainty as the value α_{kn} to be used by the conjugate gradient method, then we can calculate the number of iterations n required to achieve a desired accuracy after deciding upon an upper bound b_0. The upper bound and number of iterations used for the neural network problem will be presented in the next chapter.

5. Calculation of the Deflection Coefficient β_k

The equation used to calculate the deflection constant β_k (equation 3.3) is the Polak-Ribiere version of the conjugate gradient method originally proposed by Fletcher and Reeves [Ref. 14:p. 253]. The original method used the equation

$$\beta_k = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}. \tag{3.10}$$

to calculate the deflection constant β_k. The two equations are equivalent if the objective function to be minimized is quadratic. Experimental results, however, tend to indicate that the Polak-Ribiere method is more effective for nonquadratic objective functions [Ref. 14:p. 254]. This is because the Polak-Ribiere method tends to reset the the direction vector $\mathbf{d}_{k+1}$ to the value of the gradient vector $\mathbf{g}_{k+1}$ when two successive gradients $\mathbf{g}_k$ and $\mathbf{g}_{k+1}$ are equal. This has the effect of beginning the conjugate gradient method anew, using the present coefficients vector $\mathbf{h}_k$ as the new initial coefficient vector $\mathbf{h}_0$.

B. APPLYING THE CONJUGATE GRADIENT METHOD TO A NEURAL NETWORK

1. The Neural Network Model and Notation

The generic neural network model to be used for the purposes of discussion is pictured in Figure 3.4. The notation used when referring to the various variables of the model is as follows:

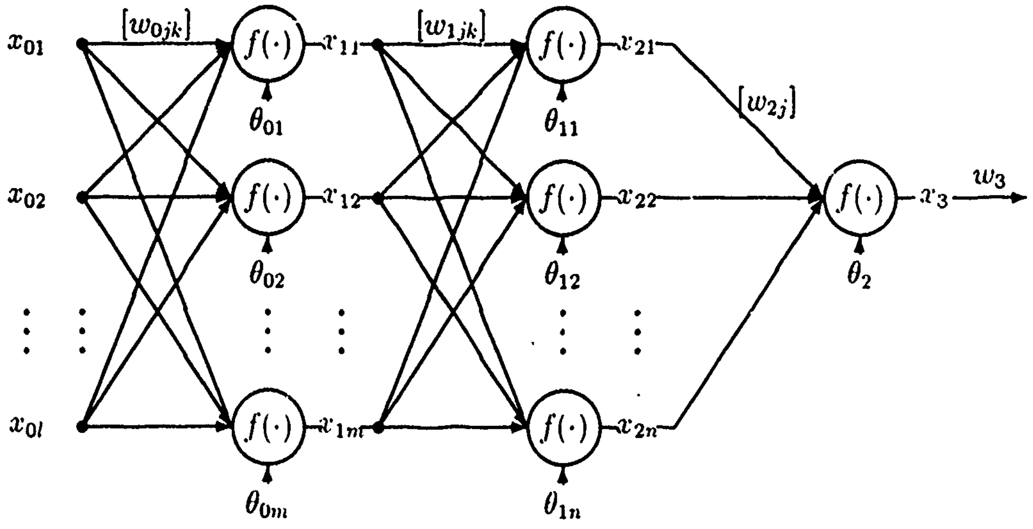

Figure 3.4: Neural network model

x_{ij}	The j^{th} input to the i^{th} layer of the network. For other than the inputs $x_{01}, x_{02}, \ldots, x_{0l}$, the variable x_{ij} is also the output of the j^{th} neuron in the $(i-1)^{\text{th}}$ layer and is a function of the previous layer's inputs and weights and the j^{th} neuron's threshold value.
w_{ijk}	The weight in the i^{th} layer of the network that connects the j^{th} input x_{ij} to the k^{th} neuron of the layer.
θ_{ik}	The threshold value associated with the k^{th} neuron of the i^{th} layer of neurons.
y	The desired output value of the network for a given set of inputs $x_{01}, x_{02}, \ldots, x_{0l}$.
$f(\cdot)$	The transfer function of the neuron.

2. The Neural Network Objective Function $J(\mathbf{h})$

As was mentioned in the previous chapter, we wish to minimize the total sum of the squared errors over an entire training data set. As a result, the objective function $J(\mathbf{h})$ to be minimized using the conjugate gradient method is

$$E = \sum_t \frac{1}{2} e^2(t) \tag{3.11}$$

where $e(t)$ is the error between the actual and the desired outputs of the neural network for the t^{th} data set.

3. The Adaptation Coefficients h

There are two quantities that we wish to adapt in order for the neural network to consistently produce the desired output for a given input. These two quantities are the connection weights w_{ijk} of the network and the threshold values θ_{ik} associated with each neuron in the network. Together, these two sets of coefficients form the coefficient vector **h**. The conjugate gradient algorithm uses a single vector **h** to represent the coefficients which are adapted to minimize the objective function $J(\mathbf{h})$. The notation used for the neural network model, however, reflects the use of matrices $[w_{ijk}]$ for the weights and vectors $[\theta_{jk}]$ for the thresholds. This was done to simplify the identification of the various weights and thresholds. We must therefore combine and transform the weight matrices and threshold vectors into a single vector **h** in order to apply the conjugate gradient algorithm. This is done by assigning the individual weights and thresholds to a vector as shown in equation 3.12.

$$\mathbf{h} = [w_{011}, w_{012}, \ldots, w_{01m}, w_{111}, \ldots, w_3, \theta_{01}, \theta_{02}, \ldots, \theta_2]^T \tag{3.12}$$

We can perform the conjugate gradient algorithm using the vector notation and then perform a reverse transformation at the completion of the algorithm to assign the final weights and threshold values to the neural network.

4. The Gradient Vector g

The gradient vector **g** used by the conjugate gradient method is defined as

$$\mathbf{g} = \frac{\partial}{\partial \mathbf{h}} J(\mathbf{h}). \tag{3.13}$$

The gradient vector **g** for the neural network problem consists of the gradients associated with the weights and thresholds of the neural network. The gradient vector **g** would be of the form

$$\mathbf{g} = [g_{011}, g_{012}, \ldots, g_{01m}, g_{111}, \ldots, g_3, g_{\theta_{01}}, g_{\theta_{02}}, \ldots, g_{\theta_2}]^T. \tag{3.14}$$

The gradient for any particular weight or threshold of the network is calculated by taking the partial derivative of the error function E with respect to the particular weight (w_{ijk}) or threshold (θ_{ik}). For the gradient associated with a weight this would be expressed as

$$g_{ijk} = \frac{\partial E}{\partial w_{ijk}} = \frac{1}{2}\frac{\partial}{\partial w_{ijk}}\left[\sum_t e^2(t)\right] \tag{3.15}$$

and for the gradient associated with a threshold as

$$g_{ijk} = \frac{\partial E}{\partial \theta_{ik}} = \frac{1}{2}\frac{\partial}{\partial \theta_{ik}}\left[\sum_t e^2(t)\right]. \tag{3.16}$$

The partial derivative in equations 3.15 and 3.16 can be moved inside the respective summation terms resulting in the following expressions

$$g_{ijk} = \frac{\partial E}{\partial w_{ijk}} = \frac{1}{2}\sum_t \frac{\partial}{\partial w_{ijk}} e^2(t) \tag{3.17}$$

and

$$g_{ijk} = \frac{\partial E}{\partial \theta_{ik}} = \frac{1}{2}\sum_t \frac{\partial}{\partial \theta_{ik}} e^2(t). \tag{3.18}$$

The gradient for each weight w_{ijk} can therefore be expressed as the sum of the partial gradients

$$g_{ijk} = \sum_t g'_{ijk}(t) \tag{3.19}$$

where the partial gradient $g'_{ijk}(t)$ is the gradient associated with the weight w_{ijk} when evaluated for a single set of training data rather than the entire training data set. The gradients associated with the threshold values of the neural network can be expressed in a similar manner, given by

$$g_{\theta_{ik}} = \sum_t g'_{\theta_{ik}}(t). \tag{3.20}$$

For the purposes of notational brevity, we will assume that the training data set consists of only one set inputs and the associated desired output. This will allow us reduce the length of equations for the gradient by removing references to the particular element of the training set used. The reader should remember, however, that if there are s pairs of data in the training set, then the gradient is the sum of the s partial gradients as expressed in equation 3.19 and equation 3.20.

a. Neuron Transfer Function Derivative

Before delving into the derivation of the equations for the gradients of the weights and thresholds of the neural network, a few comments should be made concerning the transfer function used for the neural network model and its derivative. The transfer function to be used is the sigmoid function defined by equation 2.9

in Chapter 2. A key feature of the sigmoidal function is that its derivative can be expressed in terms of its original value by

$$\frac{\partial}{\partial x} f(x) = -f(x)\left(1 - f(x)\right). \tag{3.21}$$

The derivative of a neuron's output can thus be expressed as a function of the output of the neuron and the partial derivative of the neuron's inputs. The partial derivative of the neuron's output with respect to w_{ijk} is then given by

$$\frac{\partial x_{i+1,k}}{\partial w_{ijk}} = (-x_{i+1,k})(1 - x_{i+1,k}) \frac{\partial}{\partial w_{ijk}} \left(\theta_{ik} - \sum_p w_{ip} x_{ip} \right) \tag{3.22}$$

and

$$\frac{\partial x_{i+1,k}}{\partial \theta_{ik}} = (-x_{i+1,k})(1 - x_{i+1,k}) \frac{\partial}{\partial \theta_{ik}} \left(\theta_{ik} - \sum_p w_{ip} x_{ip} \right) \tag{3.23}$$

for the derivative with respect to θ_{ik}. Equations 3.22 and 3.23 will be used frequently to evaluate the partial derivatives of each neuron's output when deriving the equations for the gradients of the neural network.

b. Calculation of the Third Layer Gradient

The calculation of the gradients for each weight and threshold of the neural network begins at the output of the neural network where the difference between the actual network output and the desired output produces an error. This error is propagated back through the network in the form of gradients. The gradient associated with the output weight w_3 can be expressed as

$$g_3 = \frac{\partial E}{\partial w_3} = \frac{\partial}{\partial w_3} \frac{1}{2} e^2 = \frac{\partial}{\partial w_3} \frac{1}{2} \left(y - w_3 x_3 \right)^2 \tag{3.24}$$

where $w_3 x_3$ is the output of the network and y is the desired output value. Taking the partial derivative yields

$$g_3 = (y - w_3 x_3) \frac{\partial}{\partial w_3} (y - w_3 x_3) = (y - w_3 x_3)(-x_3). \tag{3.25}$$

After rearranging the terms of equation 3.25, the final form for the output weight's gradient g_3 becomes

$$g_3 = (w_3 x_3 - y)\, x_3. \tag{3.26}$$

c. Calculation of the Second Layer Gradients

Derivation of the input, first and second layer gradients is somewhat more involved than that of the third layer gradients because of the multiple neurons and weights between the error at the output and the gradient for which we are deriving an expression. The gradient equation for a weight in the second layer can be expressed as

$$g_{2j} = \frac{\partial E}{\partial w_{2j}} = (y - w_3 x_3)\, \frac{\partial}{\partial w_{2j}}\, (y - w_3 x_3)\,. \tag{3.27}$$

Of the terms evaluated by the partial derivative, only the output of the third layer neuron x_3 is affected by a variation of the second layer weight w_{2j}. The desired output y can be eliminated and the partial derivative shifted to the right of the output weight term w_3. This yields the expression

$$g_{2j} = (y - w_3 x_3)\,(-w_3)\, \frac{\partial}{\partial w_{2j}}\, (x_3)\,. \tag{3.28}$$

We can replace the partial derivative term in equation 3.28 with an equivalent expression that can be evaluated with respect to w_{2j} using equation 3.22. This results in the following expression

$$g_{2j} = \underbrace{(y - w_3 x_3)\,(-x_3)}_{g_3}\,(-w_3)\,(1 - x_3)\, \frac{\partial}{\partial w_{2j}} \left(\theta_2 - \sum_p w_{2p} x_{2p}\right). \tag{3.29}$$

Comparing the first part of equation 3.29 with equation 3.26, we find that we can replace the first two terms of equation 3.29 with the output weight's gradient g_3. After taking the partial derivative, only one term, x_{2j}, remains. The equation for the second layer weight gradient becomes

$$g_{2j} = g_3\,(-w_3)\,(1 - x_3)\,(-x_{2j}) = g_3 w_3\,(1 - x_3)\, x_{2j}. \tag{3.30}$$

We can see from equation 3.30 that the gradient g_{2j} is a function of weight w_3 that connects the neuron's output to the next layer, the gradient g_3 that is associated with the output weight, the neuron's output value x_3, and the input x_{2_j} that is applied to the weight for which we are calculating the gradient (g_{2_j}). This relationship between the inputs, outputs, weights and gradients will be found to be consistent for each of the gradients of the neural network.

Rather than starting from scratch to derive the equation for the gradient associated with the output neuron's threshold θ_2, we begin at the point where evaluation of the partial derivative with respect to θ_2 differs from that for the weight gradient g_{2_j}. The equation for the gradient of the output neuron's threshold becomes

$$g_{\theta_2} = g_3\,(-w_3)\,(1 - x_3)\,\frac{\partial}{\partial \theta_2}\left(\theta_2 - \sum_p w_{2p} x_{2p}\right). \tag{3.31}$$

Evaluation of the partial derivative yields a constant of one since none of the summation terms is a function of the threshold value θ_2. Shifting the sign term, the final form for the gradient is

$$g_{\theta_2} = -g_3 w_3\,(1 - x_3) \tag{3.32}$$

Note that the equation for the gradient of the neuron's threshold value θ_2 (equation 3.32) has the same form as that for the input weights w_{2_j} connected to the output neuron (equation 3.29) except for the input term x_{2_j}. We can treat the threshold value as a weight if we assume that the threshold 'weight' has a constant input of -1.

d. Calculation of the First Layer Gradients

The derivation of the equation for the gradient of the first layer weights follows in a similar fashion to that of the second layer. We begin at the point where evaluation of the partial derivative differs (equation 3.29). The equation for the first layer weight gradient becomes

$$g_{1jk} = \frac{\partial E}{\partial w_{1jk}} = g_3\left(-w_3\right)\left(1 - x_3\right)\frac{\partial}{\partial w_{1jk}}\left(\theta_2 - \sum_p w_{2p}x_{2p}\right). \tag{3.33}$$

Only the output of the k^{th} neuron in the second layer (x_{2k}) is affected by the value of the weight w_{1jk} of the first layer. Therefore all terms except for the k^{th} term of the summation in equation 3.33 are zero when the partial derivative is taken. This yields the expression

$$g_{1jk} = -g_3w_3\left(1 - x_3\right)\left(-w_{2k}\right)\frac{\partial}{\partial w_{1jk}}x_{2k}. \tag{3.34}$$

Using equation 3.22 we can rewrite equation 3.34 as

$$g_{1jk} = \underbrace{-g_3w_3\left(1 - x_3\right)\left(-x_{2k}\right)}_{g_{2k}}\left(-w_{2k}\right)\left(1 - x_{2k}\right)\frac{\partial}{\partial w_{1jk}}\left(\theta_{1k} - \sum_q w_{1qk}x_{1q}\right). \tag{3.35}$$

The first part of equation 3.35 can be replaced with the gradient g_{2k} using equation 3.30. Only the j^{th} term of the summation under evaluation by the partial derivative with respect to w_{1jk} is nonzero. The equation for the first layer gradients of the weights then becomes

$$g_{1jk} = g_{2k}\left(-w_{2k}\right)\left(1 - x_{2k}\right)\left(-x_{1j}\right) \tag{3.36}$$

which when rearranged yields

$$g_{1jk} = g_{2k}w_{2k}\left(1 - x_{2k}\right)x_{1j}. \tag{3.37}$$

Again, the present layers's gradient is a function of the next layer's gradients and weights, the present layer's neuron output values, and the input to the present layer.

The derivation of the equation for the gradient associated with the neuron thresholds of the first layer follows in the same manner as that of the second layer. The equation for the threshold gradients θ_{1k} of the first layer can be expressed as

$$g_{\theta_{1k}} = g_{2k}\left(-w_{2k}\right)\left(1 - x_{2k}\right)\frac{\partial}{\partial \theta_{1k}}\left(\theta_{1k} - \sum_q w_{1qk}x_{1q}\right). \tag{3.38}$$

Evaluating the partial derivative results in the final equation

$$g_{\theta_{1k}} = -g_{2k}w_{2k}(1 - x_{2k}) \qquad (3.39)$$

which has the same form as equation 3.32.

e. Calculation of the Input Layer Gradients

Derivation of the input layer's gradient equation differs only slightly from the previous development. The difference is due to the fact that a variation in the value of a weight in the input layer affects the output of more than a single neuron in the next layer of the network. This means that we must retain a summation term throughout the calculation of the first layer's gradient equation. The gradient for the first layer weight can be expressed as

$$g_{0jk} = \frac{\partial E}{\partial w_{0jk}} = g_3(-w_3)(1 - x_3)\frac{\partial}{\partial w_{0jk}}\left(\theta_2 - \sum_p w_{2p}x_{2p}\right). \qquad (3.40)$$

The threshold θ_2 is not a function of the input layer's weights and is eliminated when its partial derivative is taken with respect to the input weight w_{0jk}. The other terms under evaluation by the partial derivative (i.e., x_{2p}) are all, however, a function of the input weight w_{0jk}. The partial derivative can be moved inside the summation resulting in

$$g_{0jk} = g_3w_3(1 - x_3)\sum_p w_{2p}\frac{\partial}{\partial w_{0jk}}x_{2p}. \qquad (3.41)$$

Shifting the summation to the far left and evaluating the partial derivative using equation 3.22 yields

$$g_{0jk} = \sum_p g_3w_3(1 - x_3)w_{2p}(-x_{2p})(1 - x_{2p})\frac{\partial}{\partial w_{0jk}}\left(\theta_{1p} - \sum_q w_{1qk}x_{1q}\right). \qquad (3.42)$$

The θ_{1p} term in equation 3.42 can be eliminated since it is not a function of w_{0jk}. The remaining terms can then be rearranged to produce

$$g_{0jk} = \sum_p \underbrace{g_3w_3(1 - x_3)(x_{2p})}_{g_{2p}} w_{2p}(1 - x_{2p})\frac{\partial}{\partial w_{0jk}}\sum_q w_{1qk}x_{1q}. \qquad (3.43)$$

The value g_{2p} can be substituted for the first part of equation 3.43 using equation 3.30. Also, the output of the k^{th} neuron of the input layer, x_{1k}, is a function of the input weight w_{0jk}. As a result, evaluating the partial derivative using equation 3.22 results in the equation

$$g_{0jk} = \sum_j g_{2p} w_{2p} (1 - x_{2p})\, w_{1kp} (-x_{1k})(1 - x_{1k}) \frac{\partial}{\partial w_{0jk}} \left(\theta_{0k} - \sum_r w_{0rk} x_{0r} \right). \quad (3.44)$$

Evaluating the partial derivative in equation 3.44 with respect to w_{0jk} we find that only the j^{th} term of the summation is nonzero. Rearranging the terms yields

$$g_{0jk} = \sum_p \underbrace{g_{2p} w_{2p} (1 - x_{2p})\, x_{1k}}_{g_{1kp}} w_{1kp} (1 - x_{1k})\, x_{0j}. \quad (3.45)$$

Finally, we can replace the first four terms of equation 3.45 with the value g_{1kp} using equation 3.37. This results in the equation for the gradients of the weights of the input layer of the network

$$g_{0jk} = \left(\sum_p g_{1kp} w_{1kp} \right) (1 - x_{1k})\, x_{0j}. \quad (3.46)$$

Using the same reasoning used to derive equations 3.32 and 3.39 we can express the equation for the gradient of the input layer neuron thresholds as

$$g_{\theta_{0k}} = \left(\sum_p -g_{1kp} w_{1kp} \right) (1 - x_{1k}). \quad (3.47)$$

Derivation of the equations for the gradients associated with the weights and thresholds of the neural network is now complete. What we have found is that the gradients for any particular layer of the network can be expressed as a function of the given layer's weights, thresholds, inputs, outputs, and the following layer's gradients. It is not necessary to begin at the output of the network and use the network output error $e(t)$ to calculate the gradient for a particular weight or threshold which is several layers back in the network. The above expressions for the gradient do, however, dictate that the gradient calculations begin at the output of the network and the gradients be propagated back through the network.

5. Fibonacci Line Search Parameters

Several parameters associated with the Fibonacci line search methods must be specified before the conjugate gradient algorithm described in this chapter can be applied. These parameters are:

- The initial size of the interval of uncertainty
- The number of iterations that the line search should perform.

The Fibonacci line search attempts to find the best stepsize (α_k) in which to step along the error function surface towards the global minimum in a direction defined by the direction vector ($\mathbf{d}_k$). The initial interval of uncertainty is the interval over which the algorithm will search for the optimal stepsize (α_k). The initial interval, therefore, establishes the minimum and maximum stepsize values. Our goal is to find the optimal set of weights and thresholds by moving steadily down the error function surface towards the global minimum. The lower bound of the interval, or minimum stepsize value, is therefore zero since a negative value would move the algorithm up the error function surface in a direction opposite the direction vector ($\mathbf{d}_k$). Selection of an upper bound for the interval entails a number of tradeoffs. A larger maximum value would allow the algorithm to search over a greater interval for the optimal stepsize (α_k). This could allow the conjugate gradient algorithm to converge to the global minimum more quickly by enabling it to step farther down the error function surface at each iteration of the algorithm. It could also possibly provide more protection against being trapped in a local minimum by allowing the line search algorithm to search beyond the confines of a local minimum. A larger interval, however, requires that a greater number of iterations be performed to reduce the interval of uncertainty to the required degree. This final interval of uncertainty must be small so that midpoint of the interval is reasonably close to the optimal stepsize value. It is this midpoint that is the stepsize value α_k that will be used by the conjugate gradient algorithm to update the weights and thresholds of the neural network. A larger final interval of uncertainty increases the chances of a less than optimal choice for the final stepsize. A balance must therefore be struck between the size of the initial interval of uncertainty, the size of the final interval of uncertainty, and the number of iterations to be performed.

Initial investigations were performed to determine the range of stepsize values that were typical for various neural network applications. It was found that the stepsize (α_k) generally did not exceed a value of 10.0 and was typically less than 1.0. An initial interval of uncertainty of 10.0 was therefore used throughout remainder of the thesis research.

In the course of determining the initial interval of uncertainty it was found that the line search method would occasionally yield a final step size value (α_k) which produced an error function value much greater than the previous iteration's value. It was determined that this problem was a result of the error function surface not being unimodal in the direction ($\mathbf{d}_k$) along which the algorithm searched for the minimum. If this second minimum was closer to one of the two evaluation points (λ_k and μ_k) than the true minimum, as shown in figure 3.5, then the algorithm would converge to this second minimum. This would result in an error function value larger than when the line search algorithm started. To remedy this problem, the initial interval of uncertainty was shifted to the left so that the first point evaluated was for $\lambda_0 = 0$. If the error function for the final stepsize value (α_k) was greater than the error function value with a stepsize of zero, then a stepsize of zero was returned as the final stepsize value (α_k). This had the effect of resetting the conjugate gradient algorithm. A stepsize of zero caused the algorithm to retain the same weights and thresholds for the next iteration of the algorithm. As a result, the gradient ($\mathbf{g}_{k+1}$) at the next iteration was identical to the previous gradient ($\mathbf{g}_k$) and the two successive identical gradients would produce a deflection coefficient (β_k) equal to zero. This would reset the direction vector ($\mathbf{d}_k$) to the value of the present gradient ($\mathbf{g}_k$) rather than the weighted sum of previous gradients. This had the effect of reinitializing the conjugate gradient method, but at a new starting point ($\mathbf{h}_k$) on the error function surface.

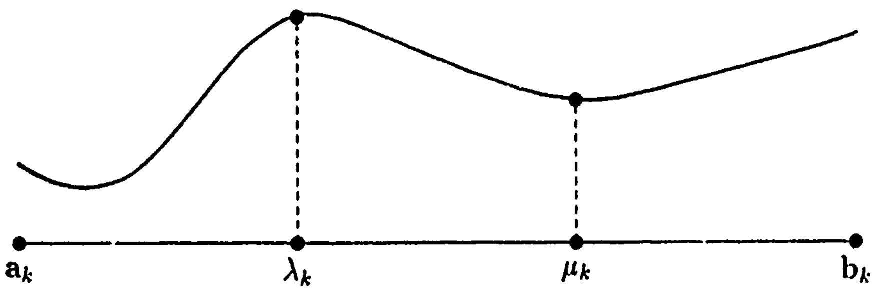

Figure 3.5: Line profile of the error function surface

Having fixed the initial interval of uncertainty, the number of iterations of the line search algorithm performed during each iteration of the conjugate gradient method was varied to determine an optimal number. Using sixteen iterations, the conjugate gradient algorithm was able to consistently reduce the value of the error function. The value of the error function did not consistently drop when fewer than sixteen iterations were used. Using equation 3.9 this resulted in a final interval of uncertainty of 0.00626.

C. COMPUTER PROGRAM IMPLEMENTATION

1. Conjugate Gradient Algorithm

The conjugate gradient algorithm was implemented for a multiple input, single output neural network using the C programming language. A flow chart showing the basic functions that are performed by the program is shown in figure 3.6. The user is prompted at the start of the program for the number of neurons in each stage of the neural network, the number of iterations of the conjugate gradient algorithm that should be performed, and the name of the input file that contains the training data that the algorithm will use to adapt the weights and thresholds of the network. The number of neurons allowed in the network is limited to a total of 50 and the number of weights connecting the neurons is limited to 500. This maximum number of neurons and weights was more than large enough for the various problems to which the program was applied. The training data file consists of columns of data in which each column is associated with an input to the neural network except for the the last column. The last column is the desired output of the neural network. Each row is a separate training data set. Upon completion of the program three files are produced. The first is a file that contains the final results. The first column of the file is the desired value and the second column is the value that the neural network produced using the final weights and thresholds of the network. If the algorithm has performed as expected and reduced the error function to a small value, then the two columns of data should be nearly identical. The second output file produced contains the final weights and thresholds of the network. This file can then be used by any other program which simulates the operation of a neural network with the same configuration of neurons. The final file is produced only if the neural network has two inputs. The file consists of a 21×21 matrix of neural network output values that were produced by applying a sequence of twenty-one evenly spaced values between 0.0 and 1.0 to each of the two inputs. The resulting file can be used to produce a three dimensional mesh of the output surface of the neural network. Examples of the input screen, output screen, and both the input and output files are contained in Appendix A. A copy of the C program source code is contained in Appendix B.

2. Backpropagation Algorithm

In order to evaluate the conjugate gradient algorithm's performance, the backpropagation method was also implemented. The basic flow chart for the backpropagation method is shown in figure 3.7. Because of the similarity between the conjugate gradient method and the backpropagation methods, this required only a few changes to the program that implemented the conjugate gradient algorithm. These changes consisted of

- Replacing the stepsize value (α_k) calculated by the Fibonacci line search with a user specified constant referred to as the learning rate by the backpropagation method.
- Replacing the deflection coefficient (β_k) which is calculated for every iteration of the algorithm with a user-specified constant referred to as the momentum factor by the backpropagation method.
- Updating the weights and thresholds of the neural network after the application of each training data set rather than upon completion of a complete pass through the entire training data file.

The input and output files remain the same as those for the conjugate gradient version of the program.

The following chapter compares the performance of the conjugate gradient and backpropagation algorithms and also presents the results of several neural network applications.

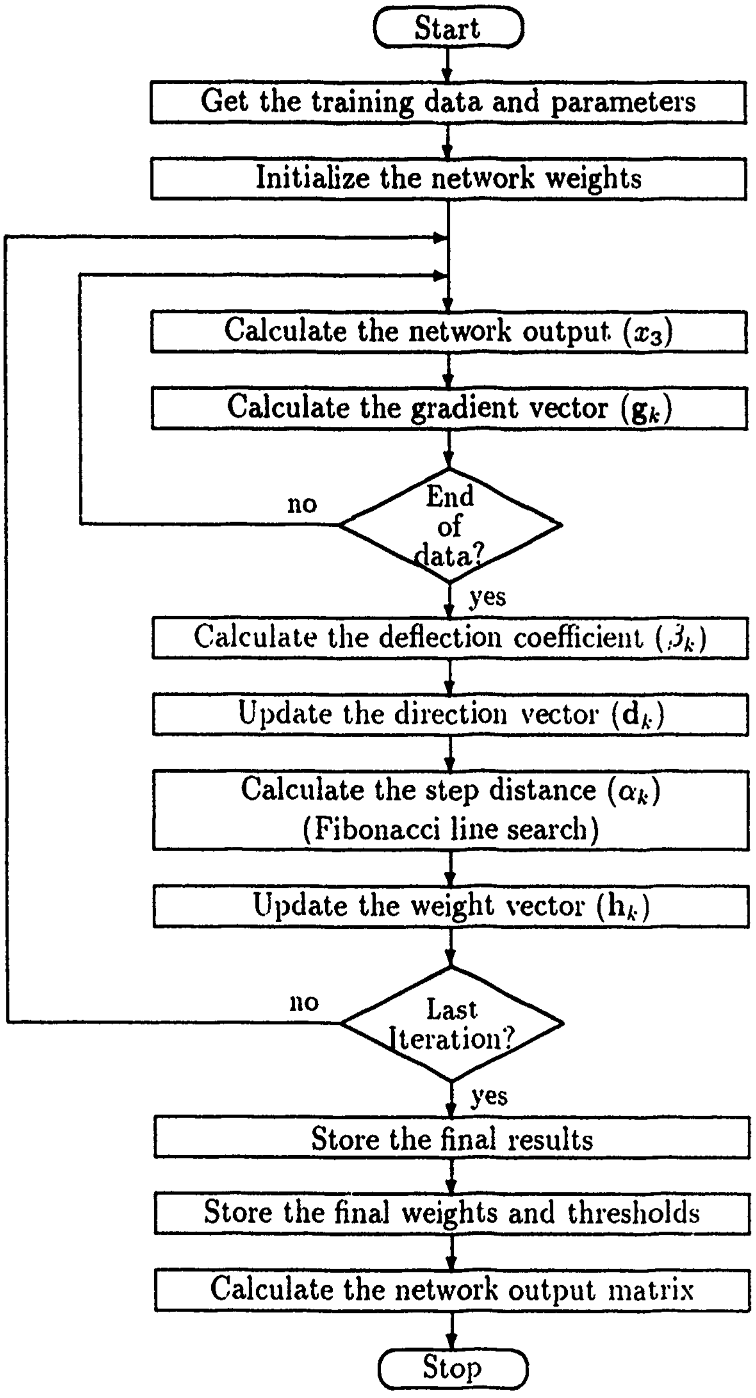
Start
Get the training data and parameters
Initialize the network weights
Calculate the network output (x_3)
Calculate the gradient vector ($\mathbf{g}_k$)
End of data?
no
yes
Calculate the deflection coefficient (β_k)
Update the direction vector ($\mathbf{d}_k$)
Calculate the step distance (α_k) (Fibonacci line search)
Update the weight vector ($\mathbf{h}_k$)
Last Iteration?
no
yes
Store the final results
Store the final weights and thresholds
Calculate the network output matrix
Stop

Figure 3.6: Conjugate gradient algorithm flowchart

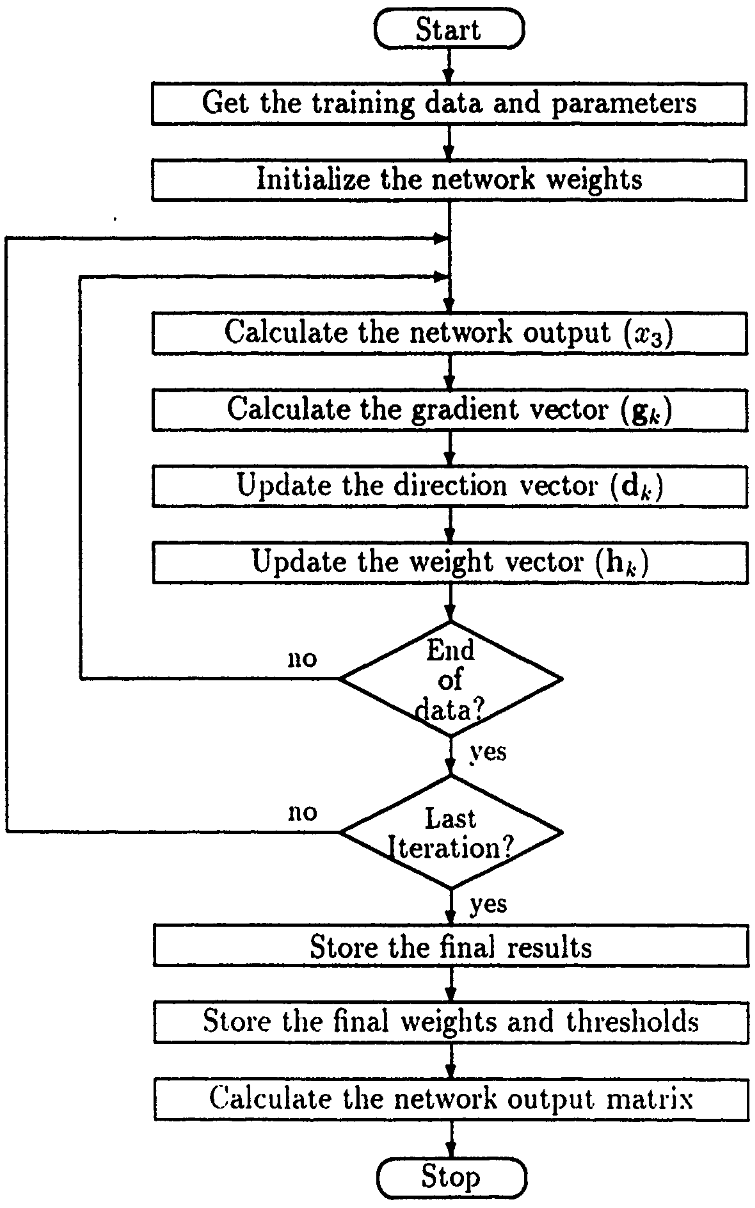
Start
Get the training data and parameters
Initialize the network weights
Calculate the network output (x_3)
Calculate the gradient vector ($\mathbf{g}_k$)
Update the direction vector ($\mathbf{d}_k$)
Update the weight vector ($\mathbf{h}_k$)
no
End of data?
yes
no
Last Iteration?
yes
Store the final results
Store the final weights and thresholds
Calculate the network output matrix
Stop

Figure 3.7: Backpropagation algorithm flowchart

IV. RESULTS

In this chapter, the results of the research conducted on neural networks using the conjugate gradient method are presented. The chapter is divided into two parts. The first concerns the performance of the conjugate gradient algorithm compared to that of the backpropagation method. The second provides several examples of neural network applications. Where possible, the performance of the neural network is compared to its linear counterpart.

A. CONJUGATE GRADIENT ALGORITHM PERFORMANCE

1. Performance Measures

The rationale for implementing the conjugate gradient algorithm was to develop an alternative to the backpropagation method that would converge more quickly to the optimal set of weights and thresholds for a given problem. The error function (E) is a measure of whether the weights and thresholds of a neural network are optimum when applied to a particular problem. The smaller the error function value, the more nearly optimum the weights and thresholds are. Both algorithms reduce the value of the error function by iteratively adapting the weights and thresholds of the neural network. The rate at which the backpropagation and conjugate gradient algorithms converge to the optimal set of weights and thresholds can be measured using several methods. The simplest approach would be to determine the number of iterations each algorithm requires to reduce the value of the error function to a prescribed level. The number of iterations for each algorithm would then be compared and the algorithm requiring fewer iterations would be considered to converge more quickly. This approach does not, however. take into account the greater computational complexity of the conjugate gradient method. A more accurate measure of performance for the purposes of comparison is the number of multiplications performed by each algorithm. This measure better reflects the relative computational requirements of the two algorithms. The number of multiplications performed by each of the methods over one iteration is fixed. We can therefore calculate a multiplication ratio of the two methods and then use this ratio in conjunction with the number of iterations to compare their relative performance.

2. Calculation of the Multiplication Ratio

The number of multiplications performed by both the backpropagation method and the conjugate gradient method over one iteration is a function of several variables. These include the number of neurons and weights in the network, the size of the training data file used to train the network, and the number of iterations performed by the Fibonacci line search method. Tables 4.1 and 4.2 show the number of multiplications required by various functions of the conjugate gradient and backpropagation method, respectively. The tables also show the total number of times each function is performed during a single iteration of the algorithm. The variable T is the number of training data sets used to train the network, the variable P is the number of weights and thresholds in the network, and R is the number of neurons in the network. Table 4.1 figures reflect that the step size (α_k) is calculated using sixteen iterations of the Fibonacci line search algorithm. The total number of multiplications (M) performed by each of the algorithms is therefore

$$M_{CG} = T(20P + 37R + 17) + 21(P + R) + 35 \tag{4.1}$$

for the conjugate gradient method and

$$M_{BP} = T(5P + 5R) \tag{4.2}$$

TABLE 4.1: MULTIPLICATIONS – CONJUGATE GRADIENT METHOD

Function	Times Performed	Number of multiplies
Calculate network output	$18T$	$P + 2R$
Calculate gradient vector	T	$2P + R$
Calculate deflection coefficient	1	$2P + 2R$
Update direction vector	1	$P + R$
Calculate step distance	1	35
Update weight vector	18	$P + R$
Calculate error sum	17	M

TABLE 4.2: MULTIPLICATIONS – BACKPROPAGATION METHOD

Function	Times Performed	Number of multiplies
Calculate network output	T	$P + 2R$
Calculate gradient vector	T	$2P + R$
Update direction vector	T	$P + R$
Update weight vector	T	$P + R$

for the backpropagation method. We can then derive the multiplication ratio by dividing M_{CG} by M_{BP} to obtain

$$RATIO = \frac{M_{CG}}{M_{BP}} = \frac{T(20P + 37R + 17) + 21(P + R) + 35}{T(5P + 5R)}. \tag{4.3}$$

Equation 4.3 can then be factored into four terms as shown below

$$RATIO = 4 + \frac{17(R+1)}{5(P+R)} + \frac{21}{5T} + \frac{35}{5T(P+R)}. \tag{4.4}$$

For the purposes of approximation, the last two terms of equation 4.4 can be eliminated since the number of training data sets used to train the neural network is typically large. As the number of neurons in a network is increased, the number of connections or weights in the network increases at a much greater rate. This happens because each neuron in a given layer is connected to every neuron in the next layer of the network. As a result, the second term of equation 4.3 steadily decreases as the number of neurons is increased. The lower bound on the multiplication ratio is therefore approximately four and the upper bound can be set at approximately five for networks having more than just a few neurons.

3. Performance Results

The performance of the conjugate gradient method was compared to the performance of the backpropagation method using two different training problems. The first consisted of training the neural network to produce a binary output of either one or zero depending on the inputs to the network. The second problem involved training the neural network to produce a specific value within the range of zero to one for a given set of inputs to the network.

A plot of the normalized value error function versus the number of iterations performed for the binary problem is pictured in Figure 4.1 for the backpropagation algorithm and in Figure 4.2 for the conjugate gradient algorithm. Note the difference in the horizontal scale of the two figures. The error function steadily decreased for the conjugate gradient method while the error function actually increased for approximately the first 100 iterations of the backpropagation algorithm. Also note that the error function's rate of change was much more even for the conjugate gradient algorithm than for the backpropagation method.

In order to compare the relative performance of the two algorithms, the multiplication ratio's upper bound of five was used. Pictured in Figure 4.3 is a comparison of the two algorithms' convergence rates with respect to the approximate number of multiplications performed by each algorithm. As can be seen, for the binary case, the conjugate gradient method consistently outperformed the backpropagation method for any given number of multiplications performed.

The results were even more apparent for the continuous output problem. The backpropagation method was unable to significantly reduce the error function's value for the first 500 iterations of the algorithm as is shown in Figure 4.4. The conjugate gradient method, however, steadily reduced the value of the error function value after each iteration of the algorithm (Figure 4.5). Comparison of the convergence rates of the two methods with respect to the number of multiplications required in each case is shown in Figure 4.6. For any given number of multiplications the conjugate gradient method greatly outperformed the backpropagation method.

The conclusion from the two examples above is that the conjugate gradient method performs as well or better than the backpropagation method with respect to both the number of iterations and the number of multiplications required to reduce the error function to a desired level. The conjugate gradient method therefore satisfies one goal of this thesis which was to develop an alternative to the backpropagation method that would converge more quickly to the optimal set of weights and thresholds for any given problem.

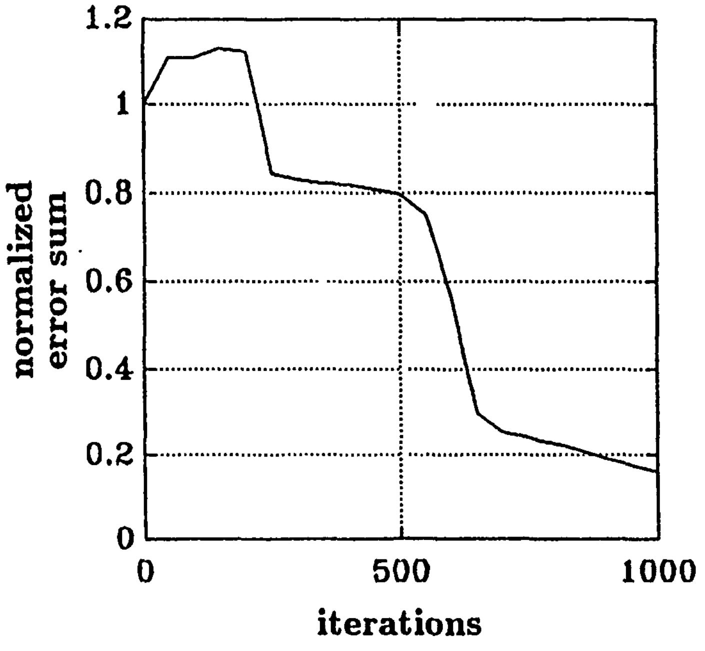

Figure 4.1: Binary problem - backpropagation

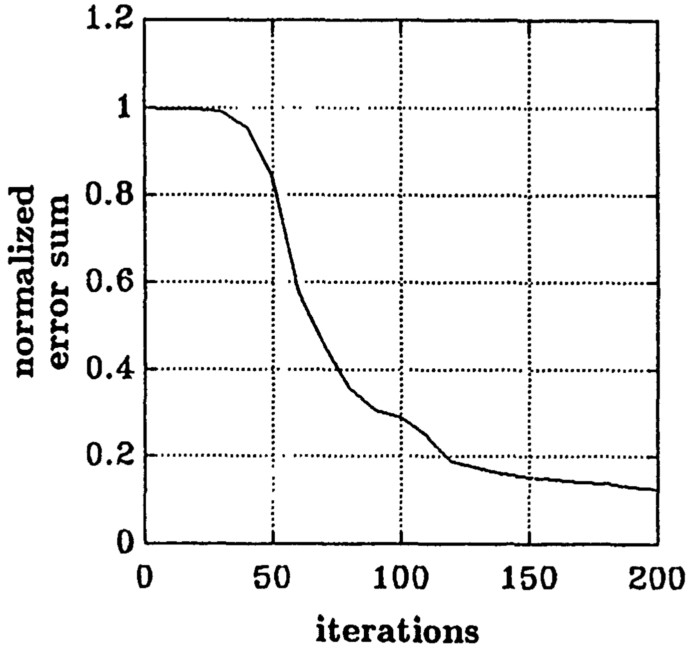

Figure 4.2: Binary problem - conjugate gradient

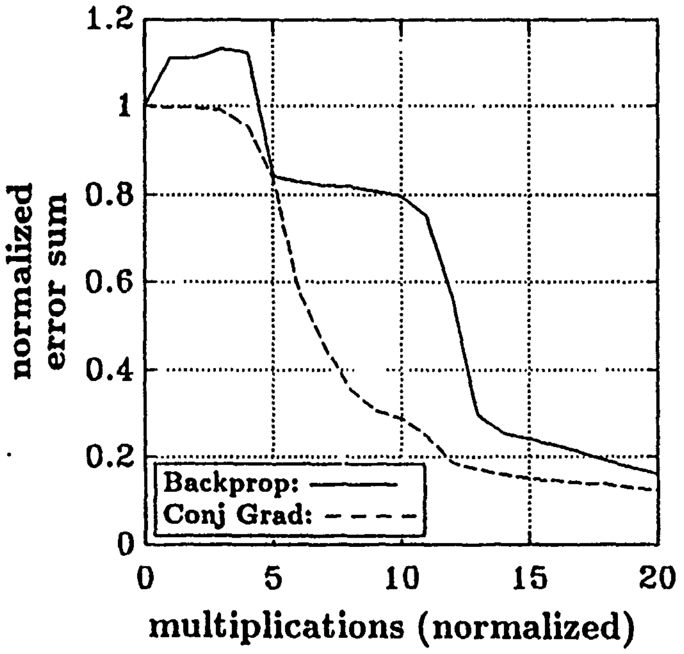

Figure 4.3: Binary problem - comparison

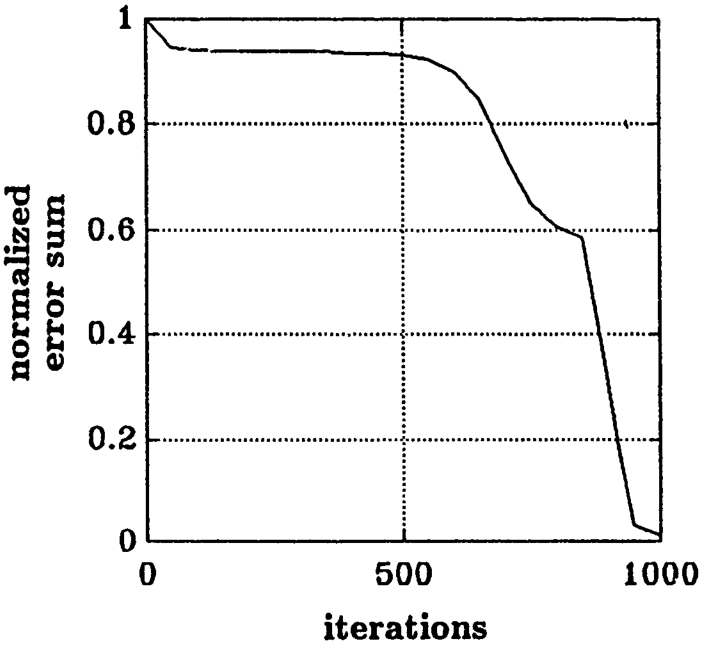

Figure 4.4: Continuous problem - backpropagation

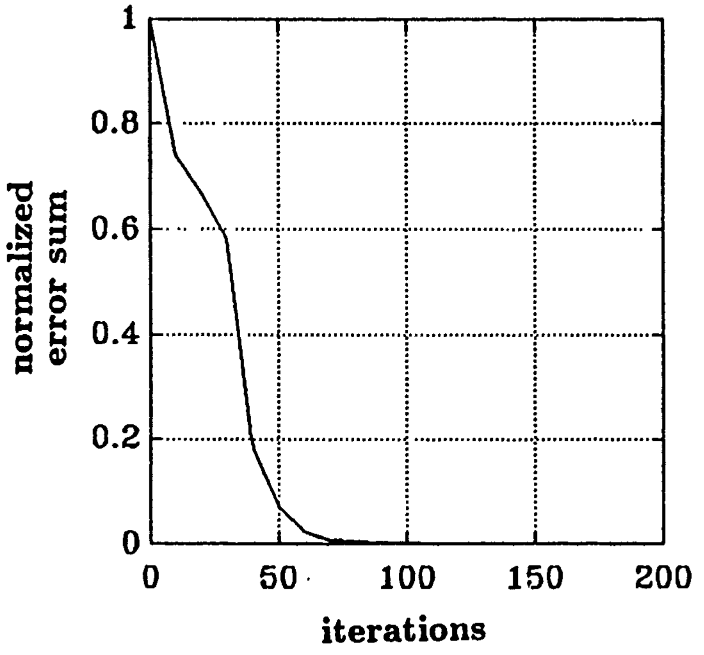

Figure 4.5: Continuous problem - conjugate gradient

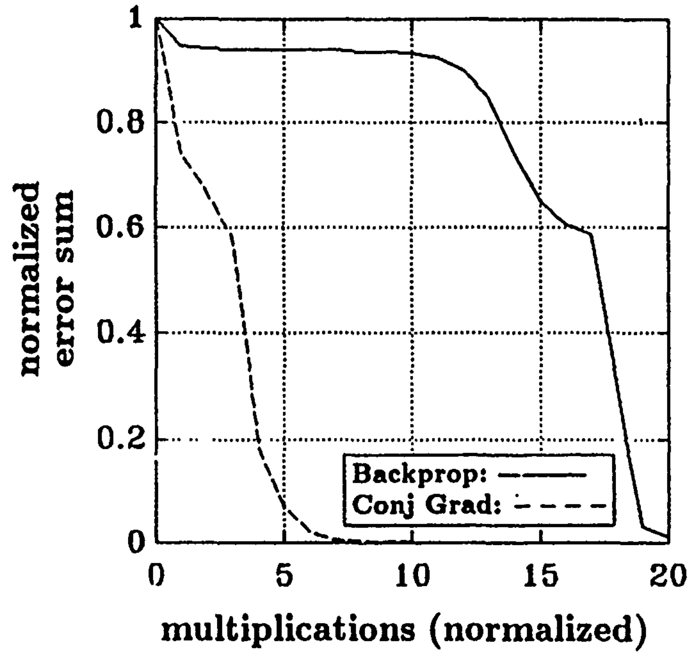

Figure 4.6: Continuous problem - comparison

B. NEURAL NETWORK APPLICATION RESULTS

Several simple applications were chosen to evaluate the performance of the conjugate gradient method vis-a-vis the backpropagation method. These applications were also used to develop a better understanding of the potential signal processing applications for the neural network. When possible, the neural network's performance was compared to its linear counterpart.

1. A Classification Problem

The goal for this problem was to train a neural network to differentiate between two classes of inputs. The two classes of inputs consisted of points which fell either inside or outside of a circle with a diameter of 0.5 centered within in a unit square as shown in Figure 4.7. This classification problem, although relatively simple, is representative of one of the primary tasks to which neural networks have been applied—pattern recognition and classification [Ref. 1:pp. 66–67].

The points used to construct the training data file were evenly spaced 0.1 apart from zero to one for both the X_0 and X_1 coordinates as shown in Figure 4.7. This produced a total of 121 points over the unit square. The training data file was composed of 121 data sets, each set consisting of the coordinates for one of the training points and a value representing the desired class to which the point belonged. The desired value for a point falling inside the circle was a one. The desired value for a point falling outside the circle was a zero. The conjugate gradient algorithm was used to train a neural network which had two inputs, eight first layer neurons, four second layer neurons, and one output neuron (a 2–8–4–1 configuration). After 100 iterations of the algorithm, the total squared error summed over the entire 121 training data sets was 6.26×10^{-2}. The resulting output of the neural network as a function of its inputs is pictured in Figure 4.8.

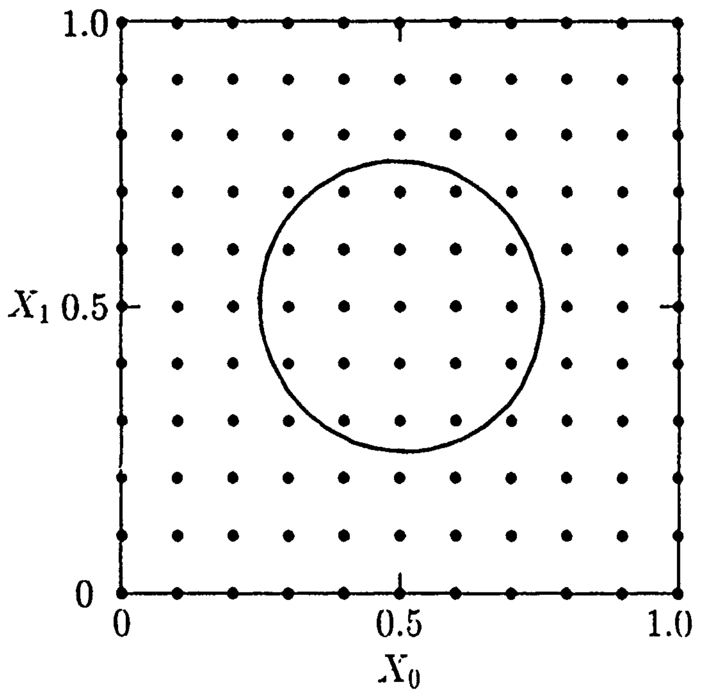

Figure 4.7: Training data for the classification problem

The neural network produced output values ranging from 1.56×10^{-9} to 1.0 and was able to properly identify the class to which each of the training data points belonged. The contour plot of the neural network output for a single contour value of 0.5 is shown in Figure 4.9. The plot clearly shows that the conjugate gradient algorithm was able to calculate a set of weights and thresholds for the neural network that very closely approximates the desired result. A circular decision region was formed that allowed the neural network to differentiate between points falling inside the circle and points falling outside the circle. This is because a neural network, due to its nonlinearity, has the ability to form arbitrarily complex decision regions.

This simple example clearly demonstrates the ability of a neural network to produce a nonlinear mapping of a set of analog inputs to a single binary output value. In this case, this nonlinear mapping was used to produce the two decision regions pictured in Figure 4.9. For other applications, the formation of decision regions may not be called for. Rather, the output of the network may have to be continuously variable.

2. Nonlinear Time Series Prediction

The previous problem required the neural network to produce only a binary output of one or zero. The second application was selected so that the conjugate gradient algorithm's performance could be evaluated for the case of a continuously variable range of desired output values. This type of application falls into a second class of tasks for which the neural network can be applied—nonlinear mapping of a set of analog inputs to an analog output value [Ref. 1:p. 67]. It was decided to apply the neural network to the problem of one-step prediction of a nonlinear time series. One-step prediction is a fairly common application in digital signal processing. A nonlinear time series was used since one-step prediction for a linear time series could easily be satisfied using a linear filter rather than a neural network. The method

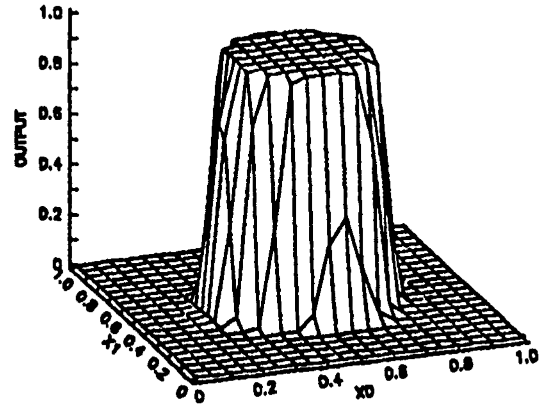

Figure 4.8: Neural network output versus input

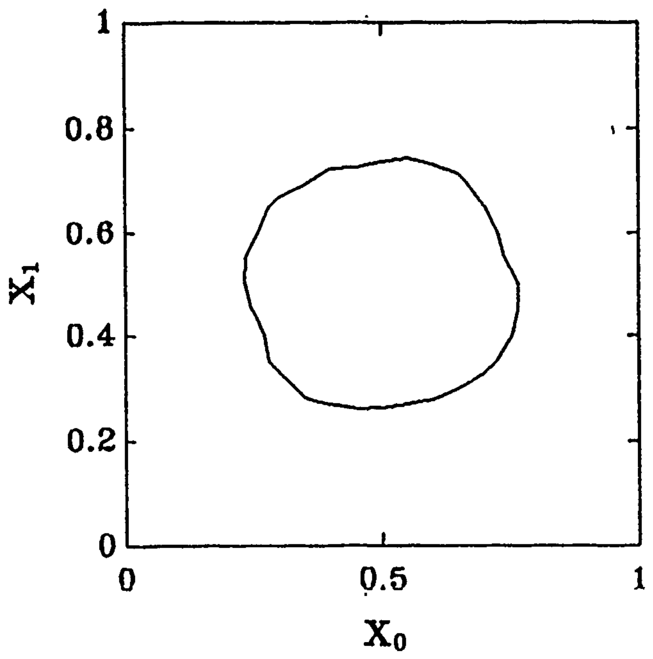

Figure 4.9: Neural network output contour plot

used to perform the prediction is similar to that used by a linear predictor. The next value in the series is predicted using the previous values of the series. The basic configuration is pictured in Figure 4.10.

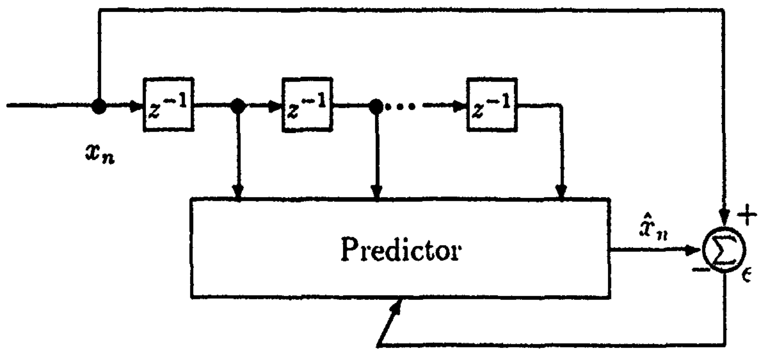

Figure 4.10: Time series predictor

For a linear predictor the output of the predictor is merely a weighted sum of a given number of previous values of the series. The neural network, however, can produce an output which is a nonlinear function of a given number of previous values. The nonlinear time series used to train and evaluate the conjugate gradient algorithm was produced using

$$x_{n+1} = 4Bx_n(1 - x_n). \tag{4.5}$$

This equation is referred to as the classic logistic or Feigenbaum map and has been studied quite extensively because its simplicity and its application to chaos theory. This iterated equation (equation 4.5) produces an ergodic, chaotic time series that is bounded and quasi–periodic [Ref. 12:p. 10]. A training sequence of 100 samples was generated using equation 4.5 with the variable B equal to 1.0. This sequence was then used to adaptively calculate the optimal coefficients for a linear second order prediction filter using a recursive least squares method. The linear predictor's results are pictured in Figure 4.11. Only the first fifty samples of the sequence were plotted so that the two curves on the graph could be better differentiated. It is obvious from Figure 4.11 that the linear predictor was unable to accurately predict the next value in the nonlinear series using the two previous values of the series. When the difference between the the actual and predicted signals is plotted one can see that the magnitude of the error is almost as great as the magnitude of the original signal (see Figure 4.12). As was expected, the linear predictor performs poorly for a nonlinear problem.

The same training sequence was then used by the conjugate gradient algorithm to train a neural network with a 2-4-2-1 configuration. The network was trained to predict the next value of the series based on the two previous values. After 100 iterations, the sum of the squared errors over the 100 training data sets was 7.25×10^{-3}. This would equate to an average standard deviation from the actual signal of approximately 8.51×10^{-3}. The neural network's results are pictured in Figure 4.13. It is apparent that the neural network performed much better than the linear predictor. The prediction error for the neural network is pictured in Figure 4.14. The magnitude of the neural network's prediction error is much smaller than that for the linear predictor. This error could also be reduced even further if additional iterations of the conjugate gradient were performed.

This example demonstrates that a neural network is quite capable of performing nonlinear mapping of a set of analog inputs to an analog output. The neural network can also produce more accurate results than the linear approach when the problem to be solved is nonlinear. It must be recognized, however, that although the neural network produces more accurate results, it is much more computationally complex than the linear approach to the problem.

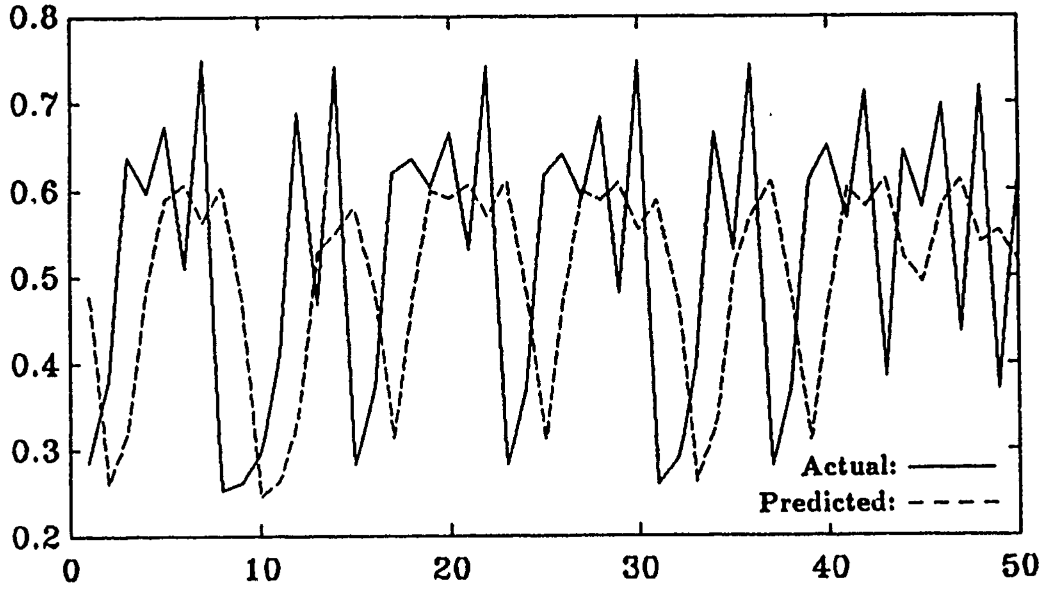

Figure 4.11: Linear predictor output and actual signal

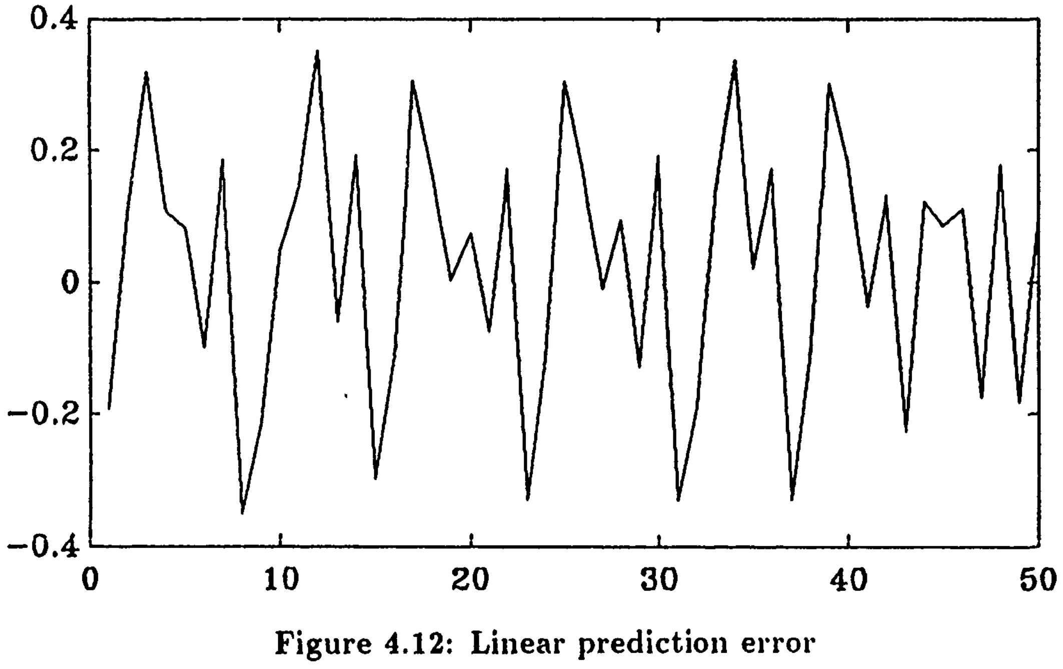

Figure 4.12: Linear prediction error

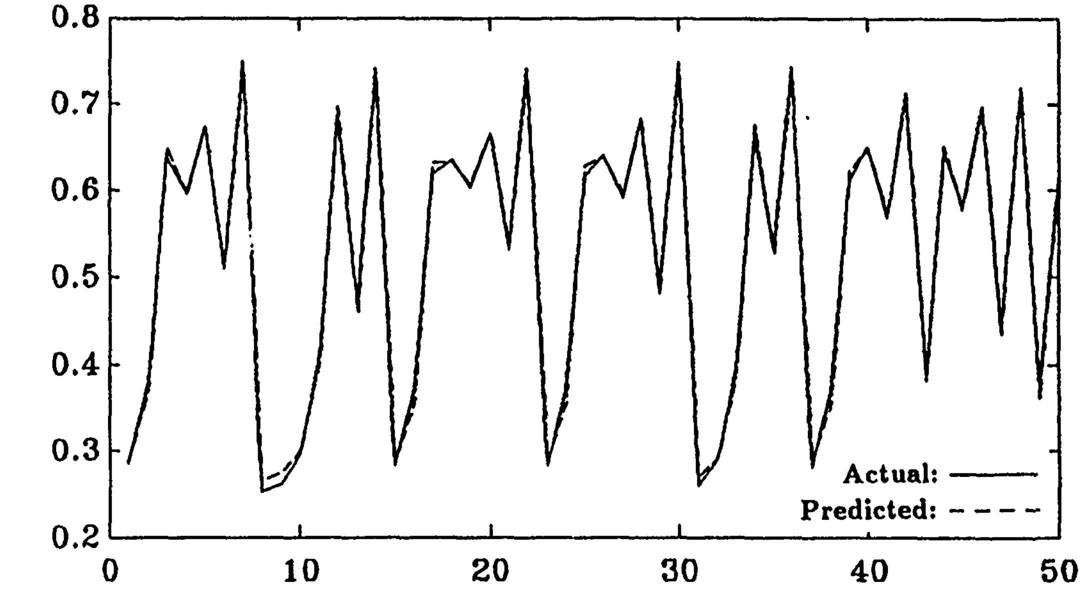

Figure 4.13: Neural network predicted and actual signal

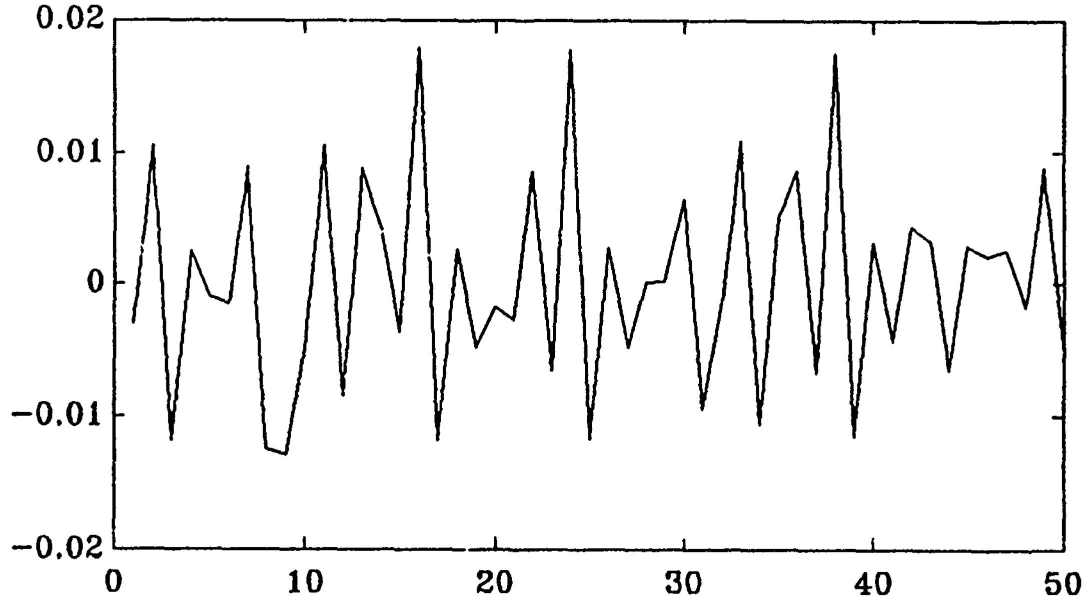

Figure 4.14: Neural network prediction error

3. Channel Equalization

One final example will serve to demonstrate the potential applications for the neural network. The idea of using a neural network to perform channel equalization for a nonminimum phase transmission channel was borrowed from Gibson, Siu, and Cowan [Ref. 15]. The experimental results indicate that a neural network could potentially provide superior performance to its linear counterpart when the channel over which the digital data is transmitted is nonminimum phase.

a. Transmission Channel Model and Equalizer model

When digital data is transmitted, it frequently becomes distorted by the channel over which it travels. This distortion can frequently be modeled using a linear time invariant (LTI) system [Ref. 8:p. 426]. The channel model, shown in Figure 4.15, consists of the transfer function $H(z)$ and a channel noise term n_i. The

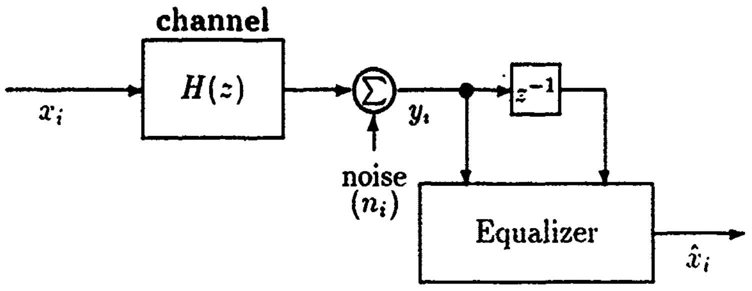

Figure 4.15: Channel model and equalizer

transfer function of the channel is defined by a finite impulse response (FIR) equation

$$H(z) = a_0 + a_1 z^{-1} + \cdots + a_k z^{-1}. \tag{4.6}$$

The channel noise term n_i is typically assumed to be zero mean, additive white gaussian noise. The purpose of a channel equalizer also shown in Figure 4.15 is to reverse the distorting effects of the channel and to recover the original signal (x_i) using

m samples of the received signal, $y_i, y_{i-1}, \ldots, y_{i-m+1}$. If we assume, for a moment, that the noise term (n_i) is zero, then the received signal y_i is merely a weighted sum of the present and past values of the original signal x_i. This can be expressed as

$$y_i = \sum_{j=0}^{k} a_j x_{i-j} \tag{4.7}$$

where a_j are the $k+1$ coefficients associated with the channel transfer function $H(z)$. For a binary signal(±1), therefore, the received signal y_i can assume only one of 2^k possible values. If we then try to estimate the original signal x_i using an m sample vector $[y_i, y_{i-1}, \ldots, y_{i-m+1}]$, we can only form a fixed number of permutations of the received signal vector. Each received signal vector $[y_i, y_{i-1}, \ldots, y_{i-m+1}]$ belongs to either the set of vectors corresponding to a transmitted binary one $(+1)$ or the set of vectors corresponding to a transmitted binary zero (-1). The channel equalizer produces an estimate of the transmitted signal x_i by determining which set the received signal vector belongs to. It has been shown that a linear transversal equalizer can perform such an operation if the channel transfer function $H(z)$ is minimum phase [Ref. 15:p. 1184]. If the channel transfer function is not minimum phase, then the two received vector sets are not linearly separable and a linear equalizer cannot accurately estimate x_i based on the received data vector set $[y_i, y_{i-1}, \ldots, y_{i-m+1}]$. If a delay, d, is introduced in the calculation of x_i, such that the at the i^{th} iteration the equalizer estimates the original signal x_{i-d}, then accurate estimation of the original signal can be achieved [Ref. 15:p. 1184]. This value for d however, may not be known, or may vary with time. The result is that a linear transversal equalizer, even with a delay, may not be able to satisfactorily equalize a nonminimum phase channel.

b. A Nonminimum Phase Channel Equalizer

The ability of a neural network to form arbitrary decision regions, demonstrated in Chapter II, could possibly remedy this problem. To investigate this concept, the first order nonminimum phase transfer function ($H(z) = 0.5 + z^{-1}$) was used to evaluate the performance of both a neural network and a linear transversal equalizer. The possible values for y_i using this transfer function are: $+1.5, +0.5, -0.5$, and -1.5. A two input neural network and two input linear transversal equalizer were used since the channel's transfer function was only first order, and this allowed a graphical analysis of the problem. The eight possible combinations of y_i and y_{i-1} are shown in Figure 4.16. The symbol $\times$ indicates that the original signal x_i had a value of -1 and the symbol $\circ$ indicates that x_i was equal to $+1$. Notice that the symbols are intermixed such that no single line can be drawn that will completely separate the two classes of symbols. This is what makes the nonminimum phase case intractable for the linear transversal equalizer. If the noise term, n_i, is now incorporated into the problem, the result is as shown in Figure 4.17 for a signal-to-noise ratio (SNR) of 10 dB. The number of possible values for y_i becomes infinite, but the points are distributed about the original eight points shown in Figure 4.16. The coefficients for a first order linear transversal equalizer were calculated by applying a recursive least squares (RLS) algorithm to the set of 500 consecutive values of y_i pictured in Figure 4.17. The values for y_i were generated by using a random sequence of $+1$ and -1 for x_i, applying this binary sequence to the transfer function given above, and adding a normally distributed noise term with a standard deviation equivalent to a signal-to-noise ratio of 10 dB. The linear transversal equalizer's two decision regions are pictured in Figure 4.18. The region that is shaded with dots is the area for which the linear transversal equalizer produced an estimate of $+1$ for x_i and the unshaded region where the equalizer produced an estimate of -1 for x_i. Note that the best that the linear equalizer could do was to define two decision regions such that three of the four possible points fell within the proper region. The same 500 value data set was then used to train a neural network having a 2-6-4-1 configuration. The decision regions formed by the neural network after 100 iterations of the conjugate gradient algorithm are pictured in Figure 4.19. The neural network, because of its ability to account for the nonlinearities, was able to form two separate decision regions for each of the two possible values for x_i. The four decision regions properly encompass the eight possible points associated with y_i and y_{i-1}. As a result, the total number of errors produced over the 500 value training set dropped from 151 for the linear equalizer to 65 for the neural network. The neural network's ability to form more complex decision regions allowed it to more accurately perform equalization when the transfer function was nonminimum phase.

c. A Nonminimum Phase Channel Equalizer Using a Delay

It was stated earlier that introduction of a delay d could allow the linear equalizer to more accurately perform its equalization function. Pictured in Figure 4.20 are the eight possible points associated with y_i and y_{i-1} for a delay of one sample (i.e., the estimate of x_{i-1} based on the samples y_i and y_{i-1}). The two classes of points are no longer intermixed as they were for the case of no delay. A set of coefficients for the linear equalizer can therefore be calculated that will properly separate the two sets of points. With noise added, however, the sets of points begin to intermix as shown in Figure 4.21 for a signal-to-noise ratio of 10 dB. The separation of the two classes becomes more difficult particularly for the linear equalizer which can only use a single line to define the decision boundary. The coefficients for the linear equalizer were again calculated using the RLS algorithm and the 500 values for y_i pictured in Figure 4.21. The resulting decision regions are shown in Figure 4.22. Comparison of the two decision regions with the original training data (Figure 4.21) indicates that the linear equalizer was unable to define a single line that could separate all the points into their proper regions. The linear equalizer produced a total of 19 errors over the 500 values of the training data set. The same training data

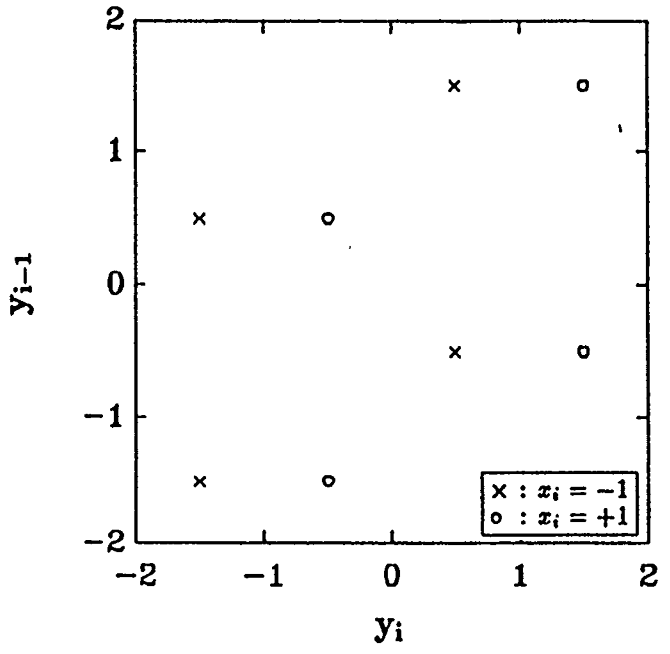

Figure 4.16: Possible combinations of y_i and y_{i-1}

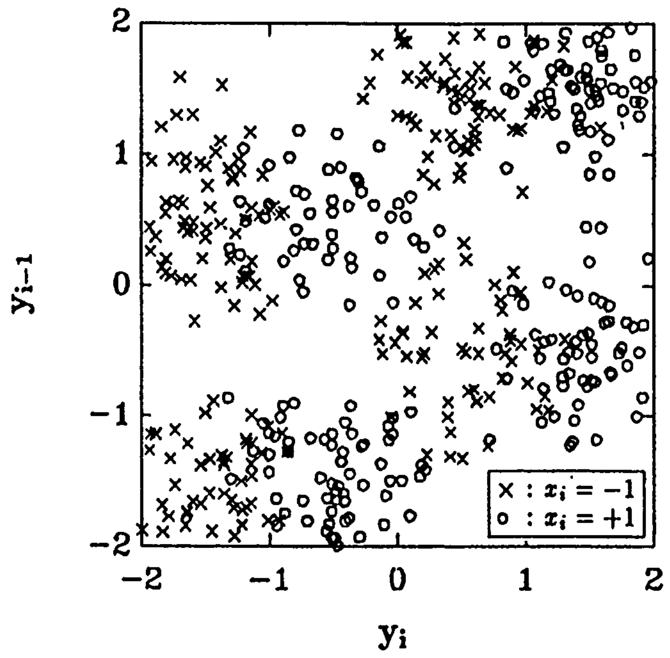

Figure 4.17: Possible combinations of y_i and y_{i-1} with noise added

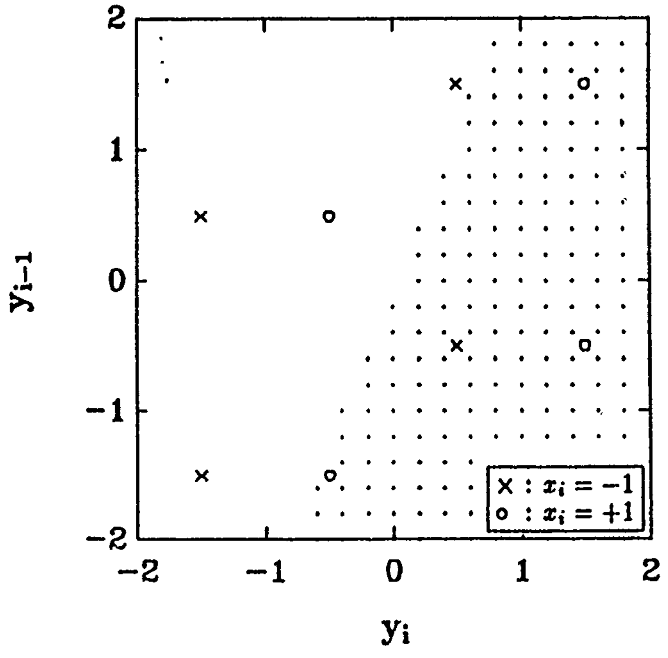

Figure 4.18: Linear equalizer decision regions

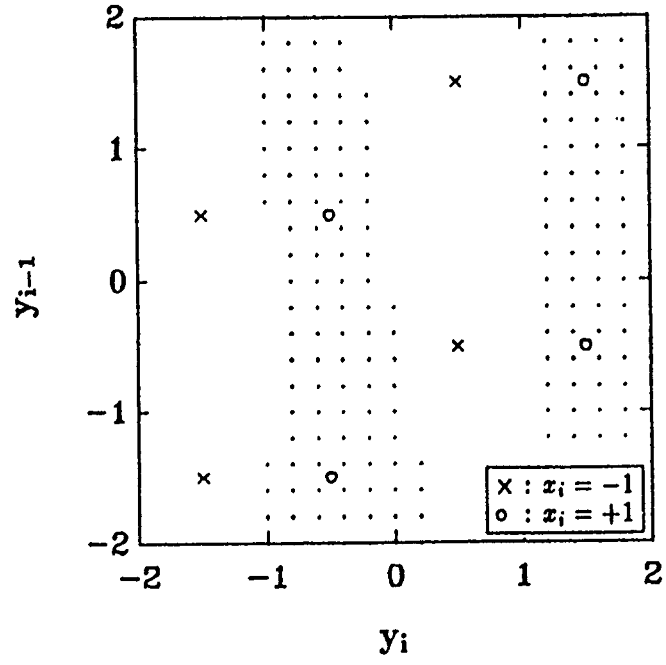

Figure 4.19: Neural network decision regions

set was then used to train a neural network with a 2-6-4-1 configuration using the conjugate gradient algorithm. After twenty iterations, the neural network produced the two decision regions pictured in Figure 4.23. The boundary between the two decision regions is no longer a straight line but is shaped to take into account the distribution of points caused by the introduction of noise. The neural network only produced a total of 3 errors over the 500 value training set.

d. A Performance Comparison

The results from the two above examples would tend to indicate that a neural network can produce superior results to the linear equalizer both when a delay is introduced and when a delay is not introduced. In order to confirm this result, the performance of both the linear transversal equalizer and the neural network were evaluated for various signal-to-noise ratios. The measure of performance for the test was the average bit error probability. The four signal-to-noise ratios: 5.0 dB, 10 dB, 20 dB, and 25 dB were used to generate four different sets of training sequences. Each sequence was generated using a different signal-to-noise ratio. Both the linear equalizer and the neural network were then trained using these four 500 value sequences for y_i. After calculating the coefficients for the linear equalizer and the weights and thresholds for the neural network the bit error performance of each type equalizer was calculated by passing the same 100,000 bit sequence through each equalizer and counting the number of times the equalizer produced an error. The results for the case where no delay was used is shown in Figure 4.24. As was expected, the bit error probability for the linear equalizer with no delay was extremely poor. The bit error probability for the neural network steadily dropped as the magnitude of the noise fell. The lowest of the three curves shown in Figure 4.24 reflects the performance of the neural network at the various sigı.al-to-noise ratios after having been trained using the 10 dB SNR training data set. Its performance is equal to or better than the

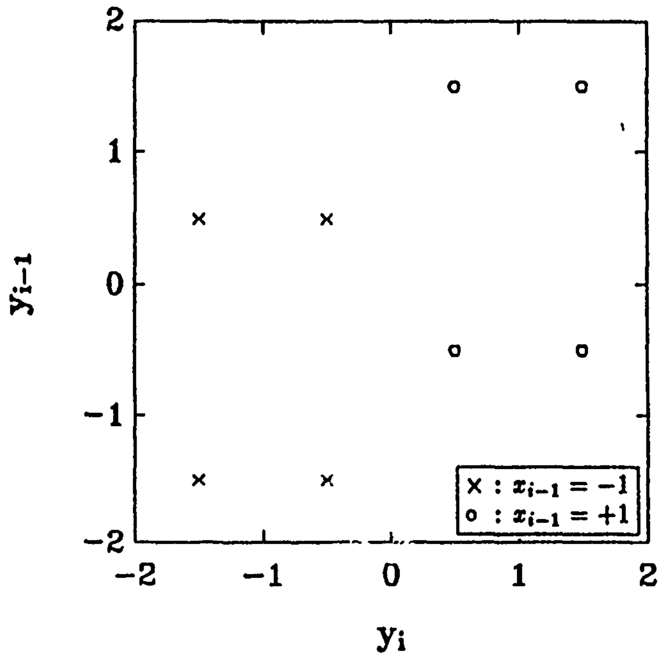

Figure 4.20: Possible combinations of y_i and y_{i-1} (with delay)

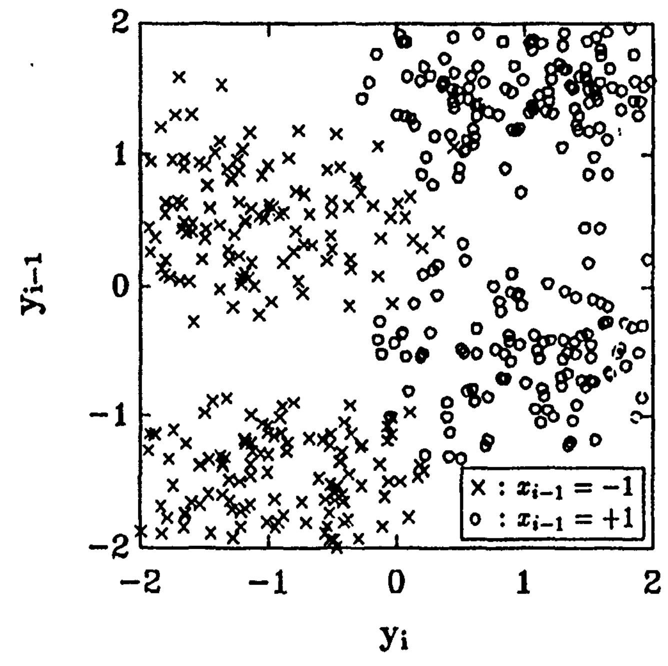

Figure 4.21: Possible combinations of y_i and y_{i-1} with noise added (with delay)

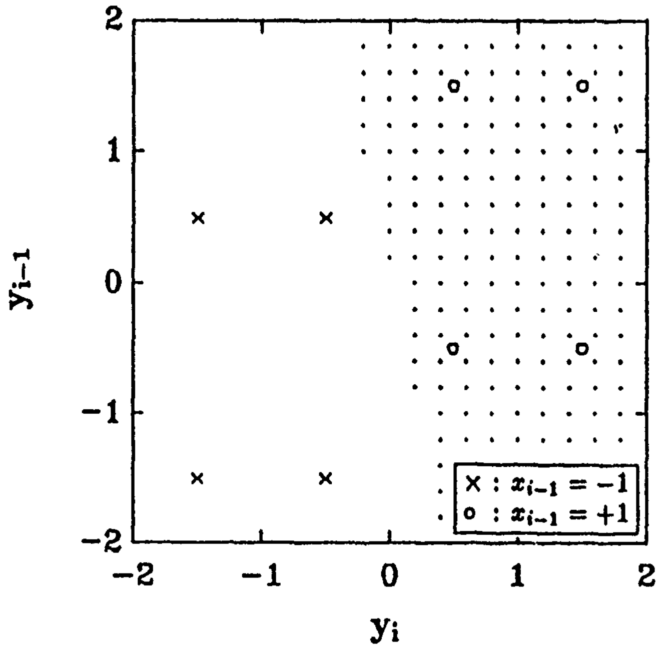

Figure 4.22: Linear equalizer (with delay) decision regions

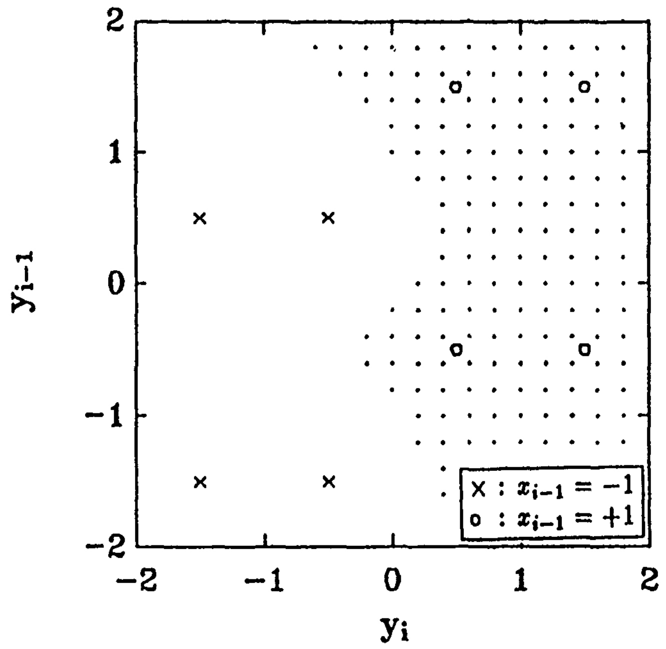

Figure 4.23: Neural network (with delay) decision regions

neural networks trained and evaluated for a specific SNR. This is because the lower SNR forced the conjugate gradient algorit^{hm} to produce a set of decision boundaries that were more nearly optimal. This res ;en more apparent for the case when a delay was introduced in the equalizatioii problem (Figure 4.25). The same method was used as described above, except that both the linear equalizer and neural network produced an estimate of x_{i-1}, rather than x_i, based on the received signals y_i and y_{i-1} . Once again the neural network performed better than the linear equalizer and the neural network trained using 10 dB data performed the best.

One final comparison can be made between the neural network and the linear transversal equalizer. This is a comparison of neural network without delay versus the linear equalizer with delay. This comparison is shown in Figure 4.26. Also shown is the neural network's performance with a delay. The neural network without delay did not perform as well as the linear equalizer for low signal-to-noise ratios. As the magnitude of the noise was reduced, however, the performance of the two approaches became comparable. The neural network with delay, however, was better than any of the approaches.

e. Channel Equalizer Conclusions

The performance of both a linear transversal equalizer and a neural network were evaluated with respect to their ability to accurately equalize a nonminimum phase digital data channel. It was found that a linear transversal equalizer was unable to accurately estimate the original signal because of the channel's nonminimum phase characteristic. The neural network, because of its ability to form arbitrary boundaries, did not suffer from this problem. Introduction of a delay allowed both the linear transversal equalizer and the neural network to improve their performance. Finally, a neural network using no delay showed a comparable performance to a linear transversal filter *with* a delay for high signal-to-noise ratios. The ability of the neural

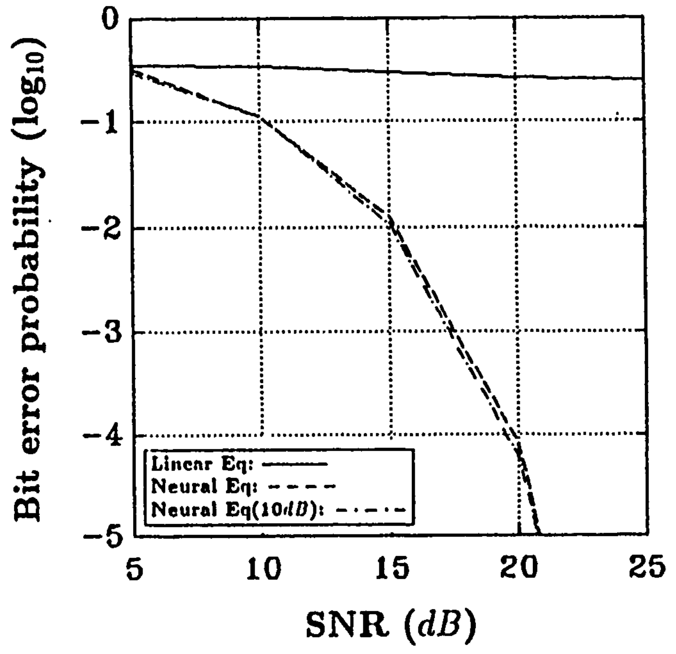

Figure 4.24: Equalizer performance (no delay)

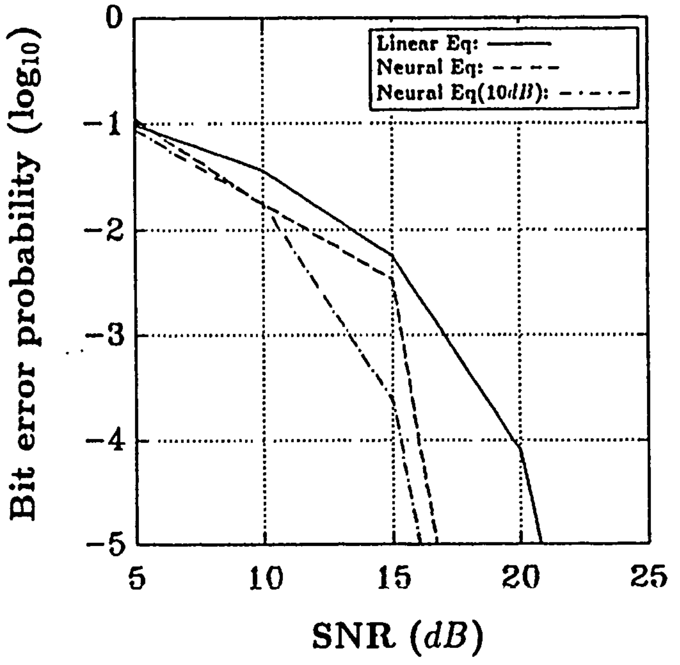

Figure 4.25: Equalizer performance (with delay)

network to perform equalization without introduction of a delay could prove useful, particularly if the required delay is unknown or varies with time.

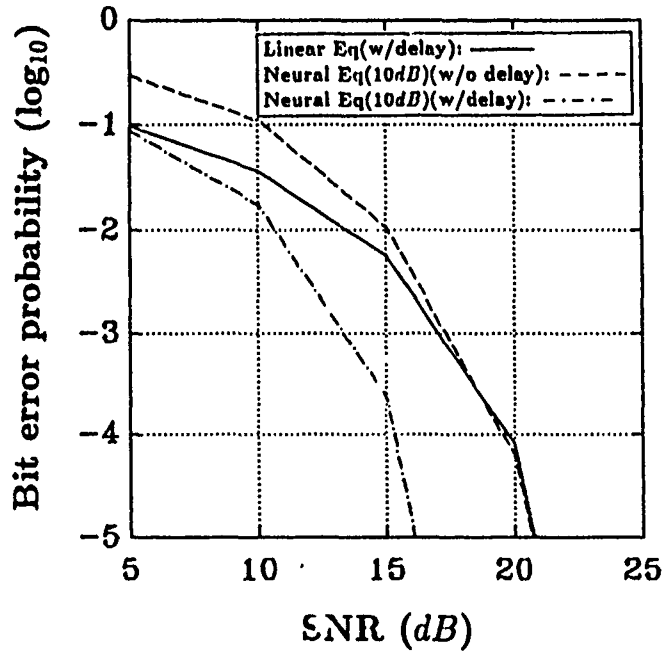

Figure 4.26: Equalizer performance – all methods

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The first objective of this thesis research was to develop an alternative to the backpropagation method for calculating the optimal set of weights and thresholds for a neural network. The results presented in Chapter IV demonstrated that the conjugate gradient algorithm developed for this thesis was more computationally efficient than the well known backpropagation method.

The second objective of this research was to develop a better understanding of the relationship between the structure of a neural network and its ability to perform input–to–output mapping. A graphical approach was used to analyze the internal representations of the neural network. The results of this analysis were presented in Chapter II.

The final objective of this thesis research was to evaluate the performance of a neural network for several different signal processing applications. The first example presented demonstrated the ability of a neural network to perform classification. The second example, nonlinear time series prediction, compared the performance of a neural network to its linear equivalent, and showed that the neural network produced superior results. The final example illustrated the performance differences between a neural network and a linear approach to nonminimum phase channel equalization.

These applications demonstrated that the nonlinear properties of a neural network frequently allow the neural network to perform functions more effectively than its linear counterpart. This is particularly the true when the problem itself is nonlinear. It must be recognized, however, that there is a cost to this increased functionality.

Calculation of the proper weights and thresholds for a given problem is much more computationally complex. The computational complexity associated with the use of a neural network must therefore be balanced with the accuracy desired when deciding whether to use a neural network rather than a linear approach to solve a given problem.

B. FUTURE RESEARCH

In the course of this thesis research, several other areas were identified that merit additional study.

1. Transfer Function Selection

The sigmoid function used for this research produced an output that ranged between 0 and 1. Other transfer functions could be investigated that produce a bipolar output. This could prove to be more useful for typical signal processing applications. One such transfer function that could be evaluated is the hyperbolic tangent function

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{2z}}. \tag{5.1}$$

This nonlinear function produces a value which ranges between ± 1 and is continuously differentiable for all values of z.

2. Neural Network Dynamic Range

The performance of a neural network having a greater dynamic range could be investigated. The dynamic range of the neural network could be expanded by allowing adaptation of the output weight w_3. It could also be accomplished by using a linear transfer function for the single neuron in output layer of the network. The output of the network would then be a linear combination of the weighted outputs from the second layer of network. This is approach taken by Lapedes and Farber in their research [Ref. 11].

3. Internal Representations

This thesis made no attempt to analyze the internal representations used by the neural network to produce the desired outputs for a given set of inputs. Research could be conducted to try to determine exactly what type of functions the individual neurons in the network perform. This could provide further insight into the relationship between the structure of a neural network and its ability to perform a particular task.

4. Analysis of the Weights and Thresholds

Research could be performed to determine if there is any analytical significance to the final weight and threshold values for a neural network.

APPENDIX A: PROGRAM OUTPUT SCREEN AND DATA FILES

A. EXAMPLE OUTPUT SCREEN

```
** Conjugate Gradient Algorithm **

What is the name of the training data file?  circ.dat
How many inputs to the neural network?  2
How many 1st layer neurons?  4
How many 2nd layer neurons?  2
There will be only one 3rd layer neuron.
How many passes thru the training data set?  2

Initial Error sum: 40.2786

Performing iteration number 1
Beta value: 0
Alpha value: 0.10958
Error sum: 17.3579

Performing iteration number 2
Beta value: 0.00366825
Alpha value: 3.79148
Error sum: 17.3564

Final error sum: 17.3557

Where do you want the results stored? circ.res
** Calculating final results **

Where do you want the final weight/theta values stored? circ.wgt
** Storing final weight/theta values **

Where do you want the map matrix stored? circ.map
** Calculating map of network **
```

B. EXAMPLE INPUT DATA FILE

```
4.0000e-001  4.0000e-001  1.0000e+000
4.0000e-001  5.0000e-001  1.0000e+000
4.0000e-001  6.0000e-001  1.0000e+000
4.0000e-001  7.0000e-001  1.0000e+000
4.0000e-001  8.0000e-001  0.0000e+000
4.0000e-001  9.0000e-001  0.0000e+000
4.0000e-001  1.0000e+000  0.0000e+000
5.0000e-001  0.0000e+000  0.0000e+000
5.0000e-001  1.0000e-001  0.0000e+000
5.0000e-001  2.0000e-001  0.0000e+000
5.0000e-001  3.0000e-001  1.0000e+000
5.0000e-001  4.0000e-001  1.0000e+000
5.0000e-001  5.0000e-001  1.0000e+000
5.0000e-001  6.0000e-001  1.0000e+000
5.0000e-001  7.0000e-001  1.0000e+000
5.0000e-001  8.0000e-001  0.0000e+000
5.0000e-001  9.0000e-001  0.0000e+000
5.0000e-001  1.0000e+000  0.0000e+000
6.0000e-001  0.0000e+000  0.0000e+000
6.0000e-001  1.0000e-001  0.0000e+000
6.0000e-001  2.0000e-001  0.0000e+000
6.0000e-001  3.0000e-001  1.0000e+000
6.0000e-001  4.0000e-001  1.0000e+000
6.0000e-001  5.0000e-001  1.0000e+000
```

Input 1 Input 2 Desired output

C. EXAMPLE RESULTS OUTPUT DATA FILE

```
1.000000e+000    1.733739e-001
1.000000e+000    1.738492e-001
1.000000e+000    1.743229e-001
1.000000e+000    1.747937e-001
0.000000e+000    1.752599e-001
0.000000e+000    1.757203e-001
0.000000e+000    1.761736e-001
0.000000e+000    1.714368e-001
0.000000e+000    1.718988e-001
0.000000e+000    1.723659e-001
1.000000e+000    1.728365e-001
1.000000e+000    1.733092e-001
1.000000e+000    1.737825e-001
1.000000e+000    1.742548e-001
1.000000e+000    1.747247e-001
0.000000e+000    1.751906e-001
0.000000e+000    1.756512e-001
0.000000e+000    1.761052e-001
0.000000e+000    1.713874e-001
0.000000e+000    1.718452e-001
0.000000e+000    1.723086e-001
1.000000e+000    1.727760e-001
1.000000e+000    1.732460e-001
1.000000e+000    1.737171e-001
```

$\underbrace{\qquad\qquad\qquad}_{\text{Desired output}}$ $\underbrace{\qquad\qquad\qquad}_{\text{Actual output}}$

D. EXAMPLE FINAL WEIGHTS OUTPUT DATA FILE

```
           2   4   2 } Number of neurons in each layer

 1.023357   0.621861  -0.194039   0.288292 } Input weights [w0]
-0.092301  -0.595949   0.007433  -0.795105 }
-0.061310  -0.031015   0.059272   0.010853 } Input thresholds [θ0]
-0.796225  -0.522201 }
-0.618687  -0.103397 } 1st layer weights [w1]
 0.513663   0.325973 }
-0.831033   0.906917 }
-0.279913   0.139071 } 1st layer thresholds [θ1]
 0.262880 } 2nd layer weights [w2]
-0.542745 }
 1.344298 } 2nd layer threshold [θ2]
 1.000000 } Output weight [w3]
```

APPENDIX B: PROGRAM SOURCE CODE LISTING

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
/*********************************************************************/
/*  This program calculates the weights and thresholds for a        */
/*  feedforward multilayer neural network using the conjugate       */
/*  gradient optimization method.                                    */
/*********************************************************************/
/*********************************************************************/
/*                      FUNCTION DECLARATIONS                        */
/*********************************************************************/
int get_info(char filename[],int num_node[]);
int get_data(char filename[],double ts_data[],int num_inputs);
int init_weights(double *weight_ptr,int num_node[]);
int init_thetas(double *theta_ptr,int num_node[]);
void adapt_network(double weight[],double theta[],int num_node[],
     int num_weights,int num_theta,double data_array[],
     int array_size,int max_iteration);
double fire_neurons(double *activity_ptr, double *weight_ptr,
    double *theta_ptr,int num_node[]);
void calc_gradient(double activity[],double weight[],
   double theta_gradient[],double gradient[],
   int num_node[],int num_weights,int num_theta);
double calc_beta(double old_gradient[],double old_theta_gradient[],
 double new_gradient[],double new_theta_gradient[],
 int num_inputs,int num_theta);
void update_direction(double gradient[],double direction[],double beta,
     int num_intputs);
void update_weights(double weight[],double alpha,double direction[],
    int num_inputs);
double calc_alpha(double weight[],double direction[],double theta[],
   double theta_direction[],double activity[],
   double data_array[],int array_size,int num_node[],
   int num_weights,int num_theta);
void load_values(double *input_ptr,double *output_ptr,int total_num);
int fibon(int n);
```

```
void write_result(double weight[],double theta[],int num_node[],
     double ts_data[],int set_size);
void map_network(double weight[],double theta[],int num_node[]);
void store_weights(double weight[],double *theta_ptr,int num_node[]);

/**********************************************************************/
/*                          MAIN PROGRAM                              */
/**********************************************************************/
main()
{
  char filename[14];
  int max_iteration,num_node[5],num_weights,set_size,num_theta;
  double ts_data[3000],weight[400],theta[50];

  printf("\n ** Conjugate Gradient Algorithm ** \n");
  max_iteration = get_info(filename,num_node);
  set_size = get_data(filename,ts_data,num_node[0]);
  if (set_size == 0){
    exit(0);
  }
  num_weights=init_weights(weight,num_node);
  num_theta=init_thetas(theta,num_node);
  adapt_network(weight,theta,num_node,num_weights,num_theta,
       ts_data,set_size,max_iteration);
  write_result(weight,theta,num_node,ts_data,set_size);
  store_weights(weight,theta,num_node);
  if (num_node[0] == 2){
    map_network(weight,theta,num_node);
  }
exit(0);
}
/**********************************************************************/
/*                          FUNCTION GET_INFO                         */
/**********************************************************************/
int get_info(char filename[],int num_node[])
{
  int max_iteration;

  printf("\n What is the name of the training data file?  ");
  flushall();
  gets(filename);
  printf("\n How many inputs to the neural network?  ");
  scanf("%2hd",&num_node[0]);
  printf("\n How many 1st layer neurons?  ");
  scanf("%2hd",&num_node[1]);
  printf("\n How many 2nd layer neurons?  ");
  scanf("%2hd",&num_node[2]);
  printf("\n There will be only one 3rd layer neuron.  ");
  num_node[3] = 1;
  num_node[4] = 1;
  printf("\n\n How many passes thru the training data set?  ");
  scanf("%4hd",&max_iteration);
  return(max_iteration);
}
/**********************************************************************/
/*                        FUNCTION GET_DATA                           */
/**********************************************************************/
int get_data(char filename[],double ts_data[],int num_inputs)
{
  FILE *stream;
  int i,num_read;

  num_read = 0;
  if ((stream = fopen(&filename[0],"r")) != NULL){
    for (i=0;(i < 3000)&&
     (fscanf(stream,"%lg",ts_data + i)>0);i++)
      ;
    fclose(stream);
    if ((i%(num_inputs+1)) != 0){
      printf("\n\n ** Improper number of input data elements **");
      num_read = 0;
    }
    else{
      num_read = i/(num_inputs+1);
    }
  }
  else{
    printf("\n\n ** Could not find the specified file **");
  }
  return(num_read);
}
/**********************************************************************/
/*                        FUNCTION INIT_WEIGHTS                       */
/**********************************************************************/
int init_weights(double *weight_ptr,int num_node[])
{
  #define MAX_VAL 16384.0
  int num_weights,i;

  srand(1);
  num_weights = 0;
  for (i=0;i<3;i++){
    num_weights += num_node[i]*num_node[i+1];
  }
  for (i=0;i<(num_node[0]*num_node[1]);i++){
    *weight_ptr++ = (1.0 - (rand()/MAX_VAL));
  }
  for (i=0;i<(num_node[1]*num_node[2]);i++){
    *weight_ptr++ = (1.0 - (rand()/MAX_VAL));
  }
  for (i=0;i<(num_node[2]*num_node[3]);i++){
    *weight_ptr++ = (1.0 - (rand()/MAX_VAL));
  }
  *weight_ptr = 1.0;
  num_weights += 1;
  return(num_weights);
}
/*********************************************************************/
/*                       FUNCTION INIT_THETAS                        */
/*********************************************************************/
int init_thetas(double *theta_ptr,int num_node[])
{
  int num_theta,i;

  num_theta = num_node[1]+num_node[2]+num_node[3];
  for (i=0;i<num_theta;i++){
    *theta_ptr++ = 0.0;
  }
  return(num_theta);
}
/*********************************************************************/
/*                      FUNCTION ADAPT_NETWORK                       */
/*********************************************************************/
void adapt_network(double weight[],double theta[],int num_node[],
   int num_weights,int num_theta,double data_array[],
   int array_size,int max_iteration)
{
  int iteration,i,j,set_num;
  double activity[50],gradient[400],direction[400],gradient_sum[400];
  double actual_output,desired_output,alpha,beta,old_gradient_mag;
  double theta_gradient[50],theta_sum[50],theta_direction[50];
  double old_gradient_sum[50],old_theta_sum[50],error,errorsum;
  double *array_ptr;

  for (iteration=0;iteration<max_iteration;iteration++){
    for (i=0;i<num_weights;i++){
      gradient_sum[i] = 0.0;
    }
    for (i=0;i<num_theta;i++){
      theta_sum[i] = 0.0;
    }
    errorsum=0.0;
    array_ptr = data_array;
    for (set_num=0;set_num<array_size;set_num++){
      for (i=0;i<num_node[0];i++){
activity[i] = *array_ptr++;
      }
      desired_output = *array_ptr++;
      actual_output=fire_neurons(activity,weight,theta,num_node);
      error = actual_output - desired_output;
      error *= error;
      gradient[num_weights-1] = (actual_output - desired_output)*
actual_output;
      calc_gradient(activity,weight,theta_gradient,gradient,num_node,
  num_weights,num_theta);
      for (i=0;i<(num_weights-1);i++){
gradient_sum[i] += gradient[i];
      }
      for (i=0;i<num_theta;i++){
theta_sum[i] += theta_gradient[i];
      }
      errorsum += error;
    }
    printf(" Error sum: %lg \n",errorsum);
    if (iteration == 0){
      beta = 0.0;
    }
    else{
      beta = calc_beta(old_gradient_sum,old_theta_sum,gradient_sum,
       theta_sum, (num_weights-1),num_theta);
    }
    for (j=0;j<(num_weights-1);j++){
      old_gradient_sum[j] = gradient_sum[j];
    }
    for (j=0;j<num_theta;j++){
      old_theta_sum[j] = theta_sum[j];
    }
    printf("\n Performing iteration number %d \n",(iteration+1));
    printf(" Beta value: %lg \n",beta);
    update_direction(gradient_sum,direction,beta,(num_weights-1));
    update_direction(theta_sum,theta_direction,beta,num_theta);
    alpha=calc_alpha(weight,direction,theta,theta_direction,activity,
     data_array,array_size,num_node,num_weights,
     num_theta);
    printf(" Alpha value: %lg \n",alpha);
    update_weights(weight,alpha,direction,(num_weights-1));
    update_weights(theta,alpha,theta_direction,num_theta);
  }
  errorsum = 0.0;
  array_ptr = data_array;
  for (set_num=0;set_num<array_size;set_num++){
    for (i=0;i<num_node[0];i++){
      activity[i] = *array_ptr++;
    }
    desired_output = *array_ptr++;
    actual_output=fire_neurons(activity,weight,theta,num_node);
    error = actual_output - desired_output;
    error *= error;
    errorsum += error;
  }
  printf("\n Final error sum: %lg \n",errorsum);
  return;
}
/**********************************************************************/
/*                       FUNCTION FIRE_NEURONS                        */
/**********************************************************************/
double fire_neurons(double *activity_ptr,double *weight_ptr,
    double *theta_ptr,int num_node[])
{
    int layer_num,neuron_num,j;
    double   mp,*input_ptr,*output_ptr;

    input_ptr = activity_ptr;
    output_ptr = activity_ptr + num_node[0];

/* Feed input forward thru each layer of the network */
    for (layer_num=0;layer_num<3;layer_num++){
      for (neuron_num=0;neuron_num < num_node[layer_num+1];neuron_num++){
temp = 0.0;
for (j=0;j < num_node[layer_num];j++){
  temp -= (*weight_ptr++)*(input_ptr[j]);
}
temp += *theta_ptr++;
*output_ptr++ = 1.0/(1.0+exp(temp));
      }
      input_ptr += num_node[layer_num];
    }
    temp = (*input_ptr) * (*weight_ptr);
    return(temp);
}
/*******************************************************************/
/*                    FUNCTION CALC_GRADIENT                       */
/*******************************************************************/
void calc_gradient(double activity[],double weight[],
   double theta_gradient[],double gradient[],
   int num_node[],int num_weights,int num_theta)
{
  int layer_num,i,j,offset;
  double *weight_ptr,*gradient_ptr,*result_gradient_ptr;
  double *output_acty_ptr,*input_acty_ptr,temp,*theta_ptr;

  weight_ptr = &weight[num_weights-1];
  gradient_ptr = &gradient[num_weights-1];
  result_gradient_ptr = gradient_ptr - 1;
  output_acty_ptr = &activity[0] + (num_node[0]+num_node[1]+num_node[2]);
  input_acty_ptr =  output_acty_ptr - 1;
  theta_ptr = &theta_gradient[num_theta-1];

  for (layer_num = 2;layer_num>-1;layer_num--){
    for (j=0;j<num_node[layer_num + 1];j++){
      temp = 0.0;
      offset = 0;
      for (i=0;i<num_node[layer_num+2];i++){
temp += (*weight_ptr) * (*gradient_ptr);
weight_ptr -= num_node[layer_num+1];
gradient_ptr -= num_node[layer_num+1];
      }
      offset = (num_node[layer_num+2]*num_node[layer_num+1])-1;
      weight_ptr += offset;
      gradient_ptr += offset;
      temp *= (1.0 - (*output_acty_ptr--));
      for (i=0;i<num_node[layer_num];i++){
(*result_gradient_ptr--) = temp * (*input_acty_ptr--);
      }
      *theta_ptr-- = (-temp);
      input_acty_ptr += num_node[layer_num];
    }
    input_acty_ptr -= num_node[layer_num];
  }
  return;
```

```
}
/**********************************************************************/
/*                      FUNCTION UPDATE_WEIGHTS                       */
/**********************************************************************/
void update_weights(double weight[],double alpha, double direction[],
    int num_inputs)
{
  int i;

  for (i=0;i<num_inputs;i++){
    weight[i] += alpha*direction[i];
  }
  return;
}
/**********************************************************************/
/*                      FUNCTION CALC_BETA                            */
/**********************************************************************/
double calc_beta(double old_gradient[],double old_theta_gradient[],
 double new_gradient[],double new_theta_gradient[],
 int num_inputs,int num_theta)
{
  int i;
  double beta,temp1,temp2;

  temp1 = 0.0;
  temp2 = 0.0;
  for (i=0;i<num_inputs;i++){
    temp1 += ((new_gradient[i]-old_gradient[i])*new_gradient[i]);
    temp2 += (old_gradient[i] * old_gradient[i]);
  }
  for (i=0;i<num_theta;i++){
    temp1 += ((new_theta_gradient[i]-old_theta_gradient[i])*
      new_theta_gradient[i]);
    temp2 += (old_theta_gradient[i] * old_theta_gradient[i]);
  }
  beta = temp1/temp2;
  if (beta < 0.0){
    beta = 0.0;
  }
  return(beta);
}
/**********************************************************************/
/*                      FUNCTION UPDATE_DIRECTION                     */
/**********************************************************************/
void update_direction(double gradient[], double direction[],
    double beta, int num_inputs)
{
  int i;

  for (i=0;i<num_inputs;i++){
    direction[i] *= beta;
    direction[i] -= gradient[i];
  }
  return;
}
/**********************************************************************/
/*                        FUNCTION CALC_ALPHA                         */
/**********************************************************************/
double calc_alpha(double weight[],double direction[],double theta[],
   double theta_direction[],double activity[],
   double data_array[],int array_size,int num_node[],
   int num_weights,int num_theta)
{
  double a,b,lamda,mu,lamda_result,mu_result,desired_result,epsilon;
  double actual_result,test_weight[500],test_theta[50],*array_ptr;
  int i,k,set_num,max_steps;

  a = 0.0;
  b = 10.0;
  max_steps = 16;
  epsilon = 0.001;
  lamda = a+((b-a)*fibon(max_steps-2)/fibon(max_steps));
  mu = a+((b-a)*fibon(max_steps-1)/fibon(max_steps));
  a -= lamda;
  b -= lamda;
  mu -= lamda;
  lamda = 0.0;
  load_values(weight,test_weight,num_weights);
  load_values(theta,test_theta,num_theta);
  update_weights(test_weight,lamda,direction,(num_weights-1));
  update_weights(test_theta,lamda,theta_direction,num_theta);
  lamda_result = 0.0;
  array_ptr = data_array;
  for (set_num=0;set_num<array_size;set_num++){
    for (i=0;i<num_node[0];i++){
    activity[i] = *array_ptr++;
    }
    desired_result = *array_ptr++;
    actual_result=fire_neurons(activity,test_weight,test_theta,
       num_node);
```

```
      actual_result -= desired_result;
      actual_result *= actual_result;
      lamda_result += actual_result;
    }
    load_values(weight,test_weight,num_weights);
    load_values(theta,test_theta,num_theta);
    update_weights(test_weight,mu,direction,(num_weights-1));
    update_weights(test_theta,mu,theta_direction,num_theta);
    mu_result = 0.0;
    array_ptr = data_array;
    for (set_num=0;set_num<array_size;set_num++){
      for (i=0;i<num_node[0];i++){
        activity[i] = *array_ptr++;
      }
      desired_result = *array_ptr++;
      actual_result=fire_neurons(activity,test_weight,test_theta,
         num_node);
      actual_result -= desired_result;
      actual_result *= actual_result;
      mu_result += actual_result;
    }
    for (k=1;(k<(max_steps-1))&&(b>0.0);k++){
      if (lamda_result > mu_result){
        a = lamda;
        lamda = mu;
        lamda_result = mu_result;
        mu = ((b-a)/fibon(max_steps-k));
        mu *= fibon(max_steps-k-1);
        mu += a;
        load_values(weight,test_weight,num_weights);
        load_values(theta,test_theta,num_theta);
        update_weights(test_weight,mu,direction,(num_weights-1));
        update_weights(test_theta,mu,theta_direction,num_theta);
        mu_result = 0.0;
        array_ptr = data_array;
        for (set_num=0;set_num<array_size;set_num++){
  for (i=0;i<num_node[0];i++){
    activity[i] = *array_ptr++;
  }
  desired_result = *array_ptr++;
  actual_result=fire_neurons(activity,test_weight,test_theta,
     num_node);
  actual_result -= desired_result;
  actual_result *= actual_result;
  mu_result += actual_result;
```

```
      }
    }
    else{
      b = mu;
      mu = lamda;
      mu_result = lamda_result;
      lamda = ((b-a)/fibon(max_steps-k));
      lamda *= fibon(max_steps-k-2);
      lamda += a;
      load_values(weight,test_weight,num_weights);
      load_values(theta,test_theta,num_theta);
      update_weights(test_weight,lamda,direction,(num_weights-1));
      update_weights(test_theta,lamda,theta_direction,num_theta);
      lamda_result = 0.0;
      array_ptr = data_array;
      for (set_num=0;set_num<array_size;set_num++){
for (i=0;i<num_node[0];i++){
  activity[i] = *array_ptr++;
}
desired_result = *array_ptr++;
actual_result=fire_neurons(activity,test_weight,test_theta,
   num_node);
actual_result -= desired_result;
actual_result *= actual_result;
 [illegible]mda_result += actual_result;
      }
    }
  }
  if (b>[illegible]
    mu = lamda + epsilon;
    load_values(weight,test_weight,num_weights);
    load_values(theta,test_theta,num_theta);
    update_weights(test_weight,mu,direction,(num_weights-1));
    update_weights(test_theta,mu,theta_direction,num_theta);
    mu_result = 0.0;
    array_ptr = data_array;
    for (set_num=0;set_num<array_size;set_num++){
      for (i=0;i<num_node[0];i++){
activity[i] = *array_ptr++;
      }
      desired_result = *array_ptr++;
      actual_result=fire_neurons(activity,test_weight,test_theta,
 num_node);
      actual_result -= desired_result;
      actual_result *= actual_result;
      mu_result += actual_result;
    }
    if (lamda_result > mu_result){
      if ((lamda+b)> 0.0){
return((lamda+b)/2.0);
      }
      else{
return(0.0);
      }
    }
    else{
      if ((lamda+a)> 0.0){
return((lamda+a)/2.0);
      }
      else{
return(0.0);
      }
    }
  }
  else{
    return(0.0);
  }
}
/***********************************************************************/
/*                      FUNCTION LOAD_VALUES                           */
/***********************************************************************/
void load_values(double *input_ptr,double *output_ptr,int total_num)
{
int i;

  for (i=0;i<total_num;i++){
    *output_ptr++ = *input_ptr++;
  }
return;
}
/***********************************************************************/
/*                         FUNCTION FIBON                              */
/***********************************************************************/
int fibon(int n)
{
  int f0,f1,f2,k;

  f2=f1=f0=1;
  if (n < 2){
    return(1);
  }
  for (k=1;k<n;k++){
    f0 = f1 + f2;
    f2 = f1;
    f1 = f0;
  }
  return(f0);
}
/*******************************************************************/
/*                     FUNCTION WRITE_RESULT                       */
/*******************************************************************/
void write_result(double weight[],double theta[],int num_node[],
     double ts_data[],int set_size)
{
  FILE *fileptr;
  char fname[14];
  int i,set_num;
  double desired_result,result,activity[50],*array_ptr;

  printf("\n\n Where do you want the results stored? ");
  flushall();
  gets(&fname[0]);
  printf("\n ** Calculating final results ** \n");
  fileptr = fopen(&fname[0],"w");
  array_ptr = ts_data;
  for (set_num=0;set_num<set_size;set_num++){
    for (i=0;i<num_node[0];i++){
      activity[i] = *array_ptr++;
    }
    desired_result = *array_ptr++;
    result = fire_neurons(activity,weight,theta,num_node);
    fprintf(fileptr," %e    %e \n",desired_result,result);
  }
  fclose(fileptr);
return;
}
/*******************************************************************/
/*                     FUNCTION MAP_NETWORK                        */
/*******************************************************************/
void map_network(double weight[],double theta[],int num_node[])
{
  int row,col;
  double result,input1,input2,activity[50];
  FILE *fileptr;
  char fname[13];
  printf("\n\n Where do you want the map matrix stored? ");
  flushall();
  gets(&fname[0]);
  printf("\n ** Calculating map of network **\n");
  fileptr = fopen(&fname[0],"w");
  input1=input2=0.0;
  for (row=0;row<21;row++){
    for (col=0;col<21;col++){
      activity[0]=input1;
      activity[1]=input2;
      result=fire_neurons(activity,weight,theta,num_node);
      fprintf(fileptr," %e",result);
      input1 += 0.05;
    }
    fprintf(fileptr,"\n");
    input1 = 0.0;
    input2 += 0.05;
  }
  fclose(fileptr);
return;
}
/********************************************************************/
/*                      FUNCTION STORE_WEIGHTS                      */
/********************************************************************/
void store_weights(double weight[],double *theta_ptr int num_node[])
{
  int i,j,k;
  double *weight_ptr1,*weight_ptr2;
  char fname[13];
  FILE *fileptr;

  printf("\n\n Where do you want the final weight/theta values stored? ");
  flushall();
  gets(&fname[0]);
  printf("\n ** Storing final weight/theta values **\n");
  fileptr = fopen(&fname[0],"w");
  for (i=0;i<3;i++){
    fprintf(fileptr,"%4d",num_node[i]);
  }
  fprintf(fileptr,"\n");
  weight_ptr2 = weight;
  for (i=0;i<3;i++){
    weight_ptr1 = weight_ptr2;
    for (j=0;j<num_node[i];j++){
      weight_ptr1 = weight_ptr2 + j;
      for (k=0;k<num_node[i+1];k++){
fprintf(fileptr,"%10.6lf ", *weight_ptr1);
weight_ptr1 += num.node[i];
      }
      fprintf(fileptr," \n");
    }
    weight_ptr2 += (num_node[i]*num_node[i+1]);
    for (j=0;j<num_node[i+1];j++){
      fprintf(fileptr,"%10.6lf ", *theta_ptr++);
    }
    fprintf(fileptr," \n");
  }
  fprintf(fileptr,"%10.6lf \n",*weight_ptr2);
  fclose(fileptr);
  return;
}
```

REFERENCES

1. DARPA, *DARPA Neural Network Study*, AFCEA International Press, Fairfax, VA, 1988.

2. James L. McClelland and David E. Rumelhart, *Explorations in Parallel Distributed Processing*, The MIT Press, Cambridge, MA, 1988.

3. Marvin Minsky and Seymour Papert, *Perceptrons*, The MIT Press, Cambridge, MA, 1969.

4. Richard P. Lippmann, "An introduction to computing with neural nets." *IEEE ASSP Magazine*, Vol. 4, No. 2, pp. 4–22, April, 1987.

5. Philip D. Wasserman and Tom Schwartz, "Neural Networks, Part 1: What are they and why is everybody so interested in them now?," *IEEE Expert Magazine*, Vol. 2, No. 4. pp. 10–12. Winter, 1987.

6. Samuel D. Stearns. "Fundamentals of adaptive signal processing." in *Advanced Topics in Signal Processing*, Jae S. Lim and Alan V. Oppenheim, eds., pp. 246–288, Prentice Hall, Englewood Cliffs, NJ, 1988.

7. Simon Haykim, *Introduction to Adaptive Filters*, Macmillan Publishing Company, New York, NY, 1984.

8. Sophocles J. Orfanidis, *Optimum Signal Processing*, Macmillan Publishing Company, New York, NY, 1988.

9. David E. Rumelhart, James L. McClelland and the PDP Research Group,"Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations*. David E. Rumelhart and James L. McClelland. eds.. pp. 318-362. The MIT Press. Cambridge, MA. 1988.

10. G.G. Lorentz. *Approximation of Functions*, 2d ed., Chelsea Publishing Company, New York, NY, 1986.

11. Alan Lapedes and Robert Farber,"How neural networks work." Report LA-UR-88-418, Los Alamos National Laboratory, Los Alamos, NM. January 1988.

12. Alan Lapedes and Robert Farber,"Nonlinear signal processing using neural networks: prediction and system modeling." Report LA- UR-87-2662. Los Alamos National Laboratory, Los Alamos, NM, July 1987.

13. Mokhtar S. Bazaraa and C. M. Shetty. *Nonlinear Programming - Theory and Algorithms*, John Wiley & Sons. New York, NY, 1979.

14. David G. Luenberger. *Linear and Nonlinear Programming*, 2d ed.. Addison-Wesley Publishing Company, Reading. MA, 1984.

15. Gavin J. Gibson, Sammy Siu, and F. N. Cowan,"Multilayer perceptron structures applied to adaptive equalizers for data communications," *Proc. of the International Conf. on Acoustics, Speech, and Signal Processing*, pp. 1183-1186, IEEE Press, New York, Pub. No. CH-2673- 2, 1989.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Superintendent, Naval Postgraduate School Attn: Professor M. Tummala, Code 62Tu Naval Postgraduate School Monterey, California 93943-5000	3
5.	Superintendent, Naval Postgraduate School Attn: Professor C. W. Therrien, Code 62Ti Naval Postgraduate School Monterey, California 93943-5000	1
6.	Dr. R. Madan (Code 1114) Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217-5000	1
7.	Superintendent, Naval Postgraduate School Attn: Professor R. Hippenstiel, Code 62Hi Naval Postgraduate School Monterey, California 93943-5000	1

8.	Professor Will Gersch Dept. of Information and Computer Sciences University of Hawaii Honolulu, Hawaii 96822	1
9.	CPT Mark D. Baehre 730 Casanova Avenue # 15 Monterey, California 93943-5000	4